Last update: July 1, 2024 Translated From: e-maxx.ru

Balanced bracket sequences

A **balanced bracket sequence** is a string consisting of only brackets, such that this sequence, when inserted certain numbers and mathematical operations, gives a valid mathematical expression. Formally you can define balanced bracket sequence with:

- e (the empty string) is a balanced bracket sequence.
- if s is a balanced bracket sequence, then so is (s).
- if s and t are balanced bracket sequences, then so is st.

For instance $(())()$ is a balanced bracket sequence, but $())($ is not.

Of course you can define other bracket sequences also with multiple bracket types in a similar fashion.

In this article we discuss some classic problems involving balanced bracket sequences (for simplicity we will only call them sequences): validation, number of sequences, finding the lexicographical next sequence, generating all sequences of a certain size, finding the index of sequence, and generating the k-th sequences. We will also discuss two variations for the problems, the simpler version when only one type of brackets is allowed, and the harder case when there are multiple types.

Balance validation

We want to check if a given string is balanced or not.

At first suppose there is only one type of bracket. For this case there exists a very simple algorithm. Let depth be the current number of open brackets. Initially $\mathrm{depth} = 0$. We iterate over all character of the string, if the current bracket character is an opening bracket, then we increment depth, otherwise we decrement it. If at any time the variable depth gets negative, or at the end it is different from 0, then the string is not a balanced sequence. Otherwise it is.

If there are several bracket types involved, then the algorithm needs to be changed. Instead of a counter depth we create a stack, in which we will store all opening brackets that we meet. If the current bracket character is an opening one, we put it onto the stack. If it is a closing one, then we check if the stack is non-empty, and if the top element of the stack is of the same type as the current closing bracket. If both conditions are fulfilled, then we remove the opening bracket from the stack. If at any time one of the conditions is not fulfilled, or at the end the stack is not empty, then the string is not balanced. Otherwise it is.

Number of balanced sequences

Formula

The number of balanced bracket sequences with only one bracket type can be calculated using the Catalan numbers. The number of balanced bracket sequences of length $2n$ (n pairs of brackets) is:

$$\frac{1}{n+1}\binom{2n}{n}$$

If we allow k types of brackets, then each pair can be of any of the k types (independently of the others), thus the number of balanced bracket sequences is:

$$\frac{1}{n+1}\binom{2n}{n}k^n$$

Dynamic programming

On the other hand these numbers can be computed using **dynamic programming**. Let $d[n]$ be the number of regular bracket sequences with n pairs of bracket. Note that in the first position there is always an opening bracket. And somewhere later is the corresponding closing bracket of the pair. It is clear that inside this pair there is a balanced bracket sequence, and similarly after this pair there is a balanced bracket sequence. So to compute $d[n]$, we will look at how many balanced sequences of i pairs of brackets are inside this first bracket pair, and how many balanced sequences with $n-1-i$ pairs are after this pair. Consequently the formula has the form:

$$d[n] = \sum_{i=0}^{n-1} d[i] \cdot d[n-1-i]$$

The initial value for this recurrence is $d[0] = 1$.

Finding the lexicographical next balanced sequence

Here we only consider the case with one valid bracket type.

Given a balanced sequence, we have to find the next (in lexicographical order) balanced sequence.

It should be obvious, that we have to find the rightmost opening bracket, which we can replace by a closing bracket without violation the condition, that there are more closing brackets than opening brackets up to this position. After replacing this position, we can fill the remaining part of the string with the lexicographically minimal one: i.e. first with as much opening brackets as possible, and then fill up the remaining positions with closing brackets. In other words we try to leave a long as possible prefix unchanged, and the suffix gets replaced by the lexicographically minimal one.

To find this position, we can iterate over the character from right to left, and maintain the balance depth of open and closing brackets. When we meet an opening brackets, we will decrement depth, and when we meet a closing bracket, we increase it. If we are at some point meet an opening bracket, and the balance after processing this symbol is positive, then we have found the rightmost position that we can change. We change the symbol, compute the number of opening and closing brackets that we have to add to the right side, and arrange them in the lexicographically minimal way.

If we don't find a suitable position, then this sequence is already the maximal possible one, and there is no answer.

```
bool next_balanced_sequence(string & s) {
    int n = s.size();
    int depth = 0;
    for (int i = n - 1; i >= 0; i--) {
        if (s[i] == '(')
            depth--;
        else
            depth++;

        if (s[i] == '(' && depth > 0) {
            depth--;
```

```
            int open = (n - i - 1 - depth) / 2;
            int close = n - i - 1 - open;
            string next = s.substr(0, i) + ')' + string(open, '(') + string(close, ')');
            s.swap(next);
            return true;
        }
    }
    return false;
}
```

This function computes in $O(n)$ time the next balanced bracket sequence, and returns false if there is no next one.

Finding all balanced sequences

Sometimes it is required to find and output all balanced bracket sequences of a specific length n.

To generate then, we can start with the lexicographically smallest sequence $((\dots(())\dots))$, and then continue to find the next lexicographically sequences with the algorithm described in the previous section.

However, if the length of the sequence is not very long (e.g. n smaller than 12), then we can also generate all permutations conveniently with the C++ STL function `next_permutation`, and check each one for balanceness.

Also they can be generated using the ideas we used for counting all sequences with dynamic programming. We will discuss the ideas in the next two sections.

Sequence index

Given a balanced bracket sequence with n pairs of brackets. We have to find its index in the lexicographically ordered list of all balanced sequences with n bracket pairs.

Let's define an auxiliary array $d[i][j]$, where i is the length of the bracket sequence (semi-balanced, each closing bracket has a corresponding opening bracket, but not every opening bracket has necessarily a corresponding closing one), and j is the current balance (difference between opening and closing brackets). $d[i][j]$ is the number of such sequences that fit the parameters. We will calculate these numbers with only one bracket type.

For the start value $i = 0$ the answer is obvious: $d[0][0] = 1$, and $d[0][j] = 0$ for $j > 0$. Now let $i > 0$, and we look at the last character in the sequence. If the last character was an opening bracket $($, then the state before was $(i-1, j-1)$, if it was a closing bracket $)$, then the previous state was $(i-1, j+1)$. Thus we obtain the recursion formula:

$$d[i][j] = d[i-1][j-1] + d[i-1][j+1]$$

$d[i][j] = 0$ holds obviously for negative j. Thus we can compute this array in $O(n^2)$.

Now let us generate the index for a given sequence.

First let there be only one type of brackets. We will use the counter depth which tells us how nested we currently are, and iterate over the characters of the sequence. If the current character $s[i]$ is equal to $($, then we increment depth. If the current character $s[i]$ is equal to $)$, then we must add $d[2n-i-1][\text{depth}+1]$ to the answer, taking all possible endings starting with a $($ into account (which are lexicographically smaller sequences), and then decrement depth.

New let there be k different bracket types.

Thus, when we look at the current character $s[i]$ before recomputing depth, we have to go through all bracket types that are smaller than the current character, and try to place this bracket into the current position (obtaining a new balance $\mathrm{ndepth} = \mathrm{depth} \pm 1$), and add the number of ways to finish the sequence (length $2n - i - 1$, balance $ndepth$) to the answer:

$$d[2n - i - 1][\mathrm{ndepth}] \cdot k^{\frac{2n-i-1-ndepth}{2}}$$

This formula can be derived as follows: First we "forget" that there are multiple bracket types, and just take the answer $d[2n - i - 1][\mathrm{ndepth}]$. Now we consider how the answer will change if we have k types of brackets. We have $2n - i - 1$ undefined positions, of which ndepth are already predetermined because of the opening brackets. But all the other brackets ($(2n - i - 1 - \mathrm{ndepth})/2$ pairs) can be of any type, therefore we multiply the number by such a power of k.

Finding the k-th sequence

Let n be the number of bracket pairs in the sequence. We have to find the k-th balanced sequence in lexicographically sorted list of all balanced sequences for a given k.

As in the previous section we compute the auxiliary array $d[i][j]$, the number of semi-balanced bracket sequences of length i with balance j.

First, we start with only one bracket type.

We will iterate over the characters in the string we want to generate. As in the previous problem we store a counter depth, the current nesting depth. At each position, we have to decide whether to place an opening or a closing bracket. To place an opening bracket, $d[2n - i - 1][\mathrm{depth} + 1] \geq k$ must be true. If so, we increment the counter depth, and move on to the next character. Otherwise, we decrement k by $d[2n - i - 1][\mathrm{depth} + 1]$, place a closing bracket, and move on.

```
string kth_balanced(int n, int k) {
    vector<vector<int>> d(2*n+1, vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)
            d[i][j] = d[i-1][j-1] + d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }

    string ans;
    int depth = 0;
    for (int i = 0; i < 2*n; i++) {
        if (depth + 1 <= n && d[2*n-i-1][depth+1] >= k) {
            ans += '(';
            depth++;
        } else {
            ans += ')';
            if (depth + 1 <= n)
                k -= d[2*n-i-1][depth+1];
            depth--;
        }
    }
    return ans;
}
```

Now let there be k types of brackets. The solution will only differ slightly in that we have to multiply the value $d[2n - i - 1][\mathrm{ndepth}]$ by $k^{(2n-i-1-\mathrm{ndepth})/2}$ and take into account that there can be different bracket types for

the next character.

Here is an implementation using two types of brackets: round and square:

```
string kth_balanced2(int n, int k) {
    vector<vector<int>> d(2*n+1, vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)
            d[i][j] = d[i-1][j-1] + d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }

    string ans;
    int shift, depth = 0;

    stack<char> st;
    for (int i = 0; i < 2*n; i++) {

        // '('
        shift = ((2*n-i-1-depth-1) / 2);
        if (shift >= 0 && depth + 1 <= n) {
            int cnt = d[2*n-i-1][depth+1] << shift;
            if (cnt >= k) {
                ans += '(';
                st.push('(');
                depth++;
                continue;
            }
            k -= cnt;
        }

        // ')'
        shift = ((2*n-i-1-depth+1) / 2);
        if (shift >= 0 && depth && st.top() == '(') {
            int cnt = d[2*n-i-1][depth-1] << shift;
            if (cnt >= k) {
                ans += ')';
                st.pop();
                depth--;
                continue;
            }
            k -= cnt;
        }

        // '['
        shift = ((2*n-i-1-depth-1) / 2);
        if (shift >= 0 && depth + 1 <= n) {
            int cnt = d[2*n-i-1][depth+1] << shift;
            if (cnt >= k) {
                ans += '[';
                st.push('[');
                depth++;
                continue;
            }
            k -= cnt;
        }

        // ']'
        ans += ']';
        st.pop();
        depth--;
    }
    return ans;
}
```

Contributors:

jakobkogler (88.6%) sgtlaugh (4.41%) alaahmet (3.68%) adamant-pwn (2.21%) 3centroids (0.37%) gnud-gnaoh (0.37%) wikku (0.37%)