Math ∪ Code

by Sahand Saba

BLOG GITHUB ABOUT

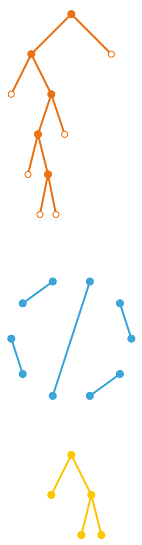
Generating All Balanced Parentheses: A Deep Dive Into An Interview Question

JUL 23, 2018

Introduction

This article is a first in a series of articles I'm planning to write that dive deeper into technical interview questions, exploring possible solutions, underlying mathematics, variations of the problem, and more, to an extent that would not often be expected during a regular technical interview (unless your interviewer happens to be Knuth). By the end of this article, we will have several algorithms to generate the following combinatorial objects in various orders:

(( ) (( ) ( )))



Wait, what are these combinatorial objects, you ask? The little animated demo above shows a visualization of all strings of length 10 consisting of balanced parentheses, all binary trees with 5 inner nodes, all non-crossing pairs of the vertices of a decagon, and all forests with 5 nodes. Those are a lot of things though, you might be thinking. True, but it turns out that they're all equivalent with relatively trivial algorithms to map from one type of object to another. We will look at some of the algorithms that do the mappings in a bit, but first let's look at the interview question that inspired the post to begin with, and a basic recursive solution for it.

Motivating Problem and Basic Solution

To get started, let's look at a technical coding question I was recently asked on a phone screen:

> *Given a positive integer n, write a program that prints all strings of length $2n$ consisting of only open and closed parentheses that are balanced.*

Balanced here means what you're likely very familiar with as a coder: every open parenthesis must have a matching closed one and they must be correctly nested. More accurately defined, a balanced string of length $2n$ contains n instances of `'('` and n instances of `')'`, and every prefix of the string must contain at least as many open parentheses as closed ones. For example, `'(())'` and `'()()'` are both balanced but `')(()'` and `'()))'` are not.

Let's look at a few examples. For $n=1$ we have just `'()'`. For $n=2$ we get two options `'(())'` and `'()()'`. For $n=3$ the following are all the possibilities:

```
()()()
()(())
(())()
(()())
((()))
```

At this point, you might observe a basic pattern: any balanced string will start with `'('`, since by definition a starting `')'` can not have a matching `'('`. Building on this observation, we can reason that the starting `'('` needs to match with a closed `')'` at some point. Hence any balanced string will be of the following format:

$$'('+x+')'+y$$

Here, x and y are themselves balanced strings. This lets us write a simple recursive solution. To do this, we do need to examine the lengths of the strings to make sure they add up correctly. We have

$$2+len(x)+len(y)=2n$$

Hence:

$$len(x)+len(y)=2n-2$$

Which means if we recursively generate all balanced strings x and y satisfying that length condition we will generate all balanced strings of length $2n$. In other words, for any $0 \leq i < n$ we can generate balanced strings of length $2i$ and $2n-2i-2$ since

$$2i + (2n-2i-2) = 2n-2$$

In other words, if we let the set of balanced strings of length $2n$ be denoted by S_n , then our recursion combines the sets S_i and S_{n-i-1} to form S_n . Let's turn this idea into Python code:

```python
def gen_balanced(n):
    if n == 0:
        yield ''
        return
    for i in range(n):
        for x in gen_balanced(i):
            for y in gen_balanced(n - i - 1):
                yield '(' + x + ')' + y
```

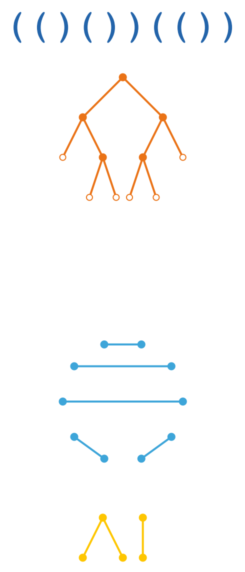
One immediate issue with the above is recursive calls for the same parameter happening multiple times. For example, calling `gen_balanced` for $n=5$ will result in 19 calls to `gen_balanced` with $n=2$. This observation, known as overlapping subproblems, means the naive recursive solution is not ideal. Fortunately, it is rather easy to fix this at the cost of the program's memory usage. Let's *memoize*! Something like the following works:

```python
table = [['']]


def gen_balanced(n):
    if n < len(table):
        return table[n]

    result = []
    for i in range(n):
        for x in gen_balanced(i):
            for y in gen_balanced(n - i - 1):
                result.append('(' + x + ')' + y)

    table.append(result)
    return table[n]
```

Seeing that, if you are at all comfortable with dynamic programming you probably have already thought of how to eliminate recursion entirely by building the table up starting from zero:

```python
def gen_balanced(n):
    table = [['']]
    for j in range(1, n + 1):
        result = []
        for i in range(j):
            for x in table[i]:
                for y in table[j - i - 1]:
                    result.append('(' + x + ')' + y)
        table.append(result)
    return table[n]
```

Before moving forward, this particular solution generates the strings in the following order:



Let's now analyze the program's run-time and space complexity. A good starting point is to simply ask how many balanced strings there are of length $2n$ given n as input. Let's denote this number by $C(n)$. That is, $C(n)=|S_n|$. Following exactly the logic of the recursion that generates the strings, we can come up with a recurrence relation for $C(n)$:

$$C(n)=\sum_{i=0}^{n-1}|S_i|\cdot|S_{n-i-1}|=\sum_{i=0}^{n-1}C(i)C(n-i-1)$$

The base case for this recurrence is $C(0)=1$ since the empty string is considered vacuously balanced. Well, now what? There's a good chance this looks familiar to you from, say, a combinatorics course. Let's assume that we do not recognize it though for the sake of learning how to deal with similar patterns in the future. It's not immediately clear looking at this recurrence if it has an easy closed form, or even what its asymptotic behaviour might be like. Let's modify our dynamic programming solution to calculate the total number of strings instead of generating the strings themselves and then list the sequence of numbers:

```
def count_balanced(n):
    table = [1]
    for j in range(1, n + 1):
        result = 0
        for i in range(j):
            x = table[i]
            y = table[j - i - 1]
            result += x * y
        table.append(result)
    return table[n]
```

The following are the first 15 values of $C(n)$:

```
1
1
2
5
14
42
132
429
1430
4862
16796
58786
208012
742900
2674440
```

While it would be fun to try to attack this sequence and try to find a closed form for it, or at least see if we can find its asymptotic behaviour, we are going to cheat a little bit and take these numbers and paste them into the Online Encyclopedia of Integer Sequences (OEIS) to see if they match a known integer sequence. The first search result on OEIS is A000108, the Catalan numbers. Wikipedia has a good article on Catalan numbers and provides the following closed form for $C(n)$:

$$C(n)=\frac{1}{n+1}\binom{2n}{n}$$

The same article provides six different proofs for this formula (yes, not kidding, *six*). It's also interesting to see how many different combinatorial objects are counted by the Catalan numbers. Richard P. Stanley in Enumerative Combinatorics Volume 2 goes over many combinatorial structures that the Catalan numbers count. See this excerpt from the book for 66 sets counted by the Catalan numbers.

The asymptotic growth of $C(n)$ is estimated by the following formula:

$$C(n) = \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

This means our algorithm uses $O(\frac{4^n}{n^{3/2}}) = O(4^n)$ memory in the memoized cased. The non-memoized version uses less memory (assuming a Python generator is used, as was the case in the solution above) but as discussed, the same sub-problems are solved again and again resulting in exponential work done *per each output string*. Of course, it is important to consider the amount of work done per item in the output, as the total number of output strings is exactly $C(n) = O(4^n)$ and hence we need to do minimum $O(C(n)) = O(4^n)$ amount of work total even if we find the most optimal solution that does constant amount of work on average per string.

A Better Solution: Generation in Lexicographic Order

Can we come up with an algorithm that generates all the output balanced strings without using exponential amounts of memory, or needing exponential time per balanced string? Our previous solutions all relied on the recursive nature of the output set (i.e., we relied on self-similarity) which led to recursive algorithms to generate the set. Instead, an alternative approach is to try to come up with a way to move from one object to the next, instead of relying on symmetric properties of the whole set. Of course this means we have to have a precise definition of the word "next". One of the simplest ways to define "next" is to define the order of the output to be lexicographic, i.e., dictionary sorting order.

Before proceeding, let's first look at what lexicographic sorted output would look like for $n=4$, where we assume $'(' < ')'$:

```
(((())))
((()()))
((())())
((()))()
(()(()))
(()()())
(()())()
(())(())
(())()()
()((()))
()(()())
()(())()
()()(())
()()()()
```

How can we move from one string to the immediate next one in lexicographic order? One possible answer is to simply start moving from right to left, and look for the first opportunity to "increment" a single character. Here, "increment" means to go from one character to the one immediately succeeding it in lexicographic order. Once such an increment is made, we then replace everything to the right of what we changed with the smallest possible lexicographic string that satisfies the property we are looking for.

This approach might sound obscure, but if you can count, you have already been doing it! Consider how you move from the number `4999` to the next which is `5000`: you start from the right and notice that you can not increment a `9` in base `10` (as a single character), so you keep moving until you get to `4` and increment the `4` to a `5` and then go back and replace everything to the right of `5` with a sequence zeros since that is the smallest lexicographic string.

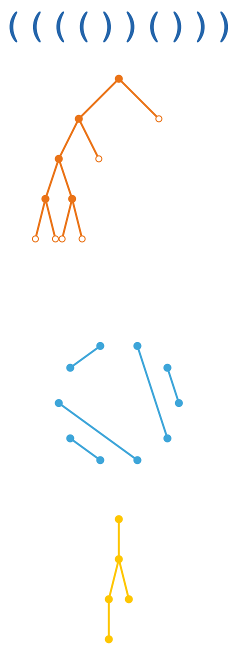
Let's apply this strategy to our balanced strings problem. Several observations:

- The only way to increment a character is changing a `'('` to a `')'`.
- We can only change a `'('` to a `')'` if doing so will not violate the definition of a balanced string. This means we can only change a `'('` to a `')'` if there are enough `')'` following it to ensure there are matching parentheses.
- Once we replace a `'('` with a `')'`, the smallest lexicographic string to replace the suffix with is `((...(())))...))` with the right number of open and closed parentheses.

Carefully considering the above observations leads to the following simple yet beautiful iterative algorithm, which repeatedly replaces the right instance of `((...(()))...)))` with `)(...(()))...)))`.

```python
def gen_balanced_iterative(n):
    # Start with the lexicographically smallest string.
    s = ['('] * n + [')'] * n
    while True:
        yield ''.join(s)
        o, c = 0, 0  # Opening and closing parentheses count
        for i in range(1, 2 * n + 1):
            # If we are checking the very first character in the string we are
            # definitely done since we can not possibly change the first
            # character from ( to ).
            if i == 2 * n:
                return
            if s[-i] == '(':
                o += 1
                if c > o:
                    # This is our opportunity to change '(' to ')' at index i
                    # and then replace the rest of the string with the smallest
                    # lexicographic suffix, which is o opening and c - 1
                    # closing parentheses (we already placed one closing
                    # parenthesis in the string so c - 1 left to place).
                    s[-i:] = [')'] + ['('] * o + [')'] * (c - 1)
                    break
            else:
                c += 1
```

What does the output of this algorithm look like? Let's visualize it:



One thing to notice, especially by looking at the visualization, is that there is a certain "jumpiness" to the output. By that I mean adjacent strings tend to at times differ from one another quite substantially. For example, consider `'((()))()()'` which is followed by `'(()((())))'`. The Hamming distance between the two is 6 which is quite large for strings of length 10. Also note that for $n=5$ as we have in the visualization, the maximum Hamming distance is 8 for balanced strings since all balanced strings start with `'('` and end with `')'`. We will revisit this issue of "jumpiness" later in the article.

What is the space complexity of this algorithm? It's pretty obvious: $O(n)$ since we just maintain a single list of length $2n$ and a few extra integers (`o` and `c` in particular). What about time complexity? It's clear that we do at most a linear amount of work per output string, since we scan from right to left once, and then from left to right another time, so time complexity per output string is $O(2n)=O(n)$.

However, this is the worst case for each string. It is more interesting to know on average how much work is done. In other words, suppose we check $t(s)$ characters starting from the right before finding an opportunity to increment a character starting from balanced string s (i.e., $t(s)$ is the number of comparisons done starting from string s). Then $t(s)$ is a good proxy for measuring the time complexity of the algorithm since it's easy to see the total actual number of operations per string is a constant multiple of $t(s)$ plus some initial overhead. We are interested in the function $A(n)$ of n defined as follows:

$$A(n)=\frac{\sum_{s\in S_n} t(s)}{C(n)}$$

Here, as you recall, S_n is the set of all balanced strings of length n , the set we are writing a program to generate. Let's let $T(n)=\sum_{s\in S_n} t(s)$. Then $A(n)=\frac{T(n)}{C(n)}$ so we are interested in calculating $T(n)$. Let's write a program to calculate the values of $T(n)$ first. Here it is:

```
def balanced_iterative_operations(n):
    T, C = 0, 0
    s = ['('] * n + [')'] * n
    while True:
        C += 1
        o, c = 0, 0
        for i in range(1, 2 * n + 1):
            if s[-i] == '(':
                o += 1
                if c > o:
                    T += i
                    s[-i:] = [')'] + ['('] * o + [')'] * (c - 1)
                    break
            else:
                c += 1
        if o == n:
            T += 2 * n
            break

    return T, C


for n in range(1, 15):
    T, C = balanced_iterative_operations(n)
    print('n = {:#2d}\tT = {:#10d}\tC = {:#10d}\tA = {:.5}'
          .format(n, T, C, T / C))
```

The output is shown below for the first 18 values:

```
n =  1      T =          2  C =          1  A = 2.0
n =  2      T =          7  C =          2  A = 3.5
n =  3      T =         21  C =          5  A = 4.2
n =  4      T =         63  C =         14  A = 4.5
n =  5      T =        195  C =         42  A = 4.6429
n =  6      T =        624  C =        132  A = 4.7273
n =  7      T =       2054  C =        429  A = 4.7879
n =  8      T =       6916  C =       1430  A = 4.8364
n =  9      T =      23712  C =       4862  A = 4.877
n = 10      T =      82498  C =      16796  A = 4.9118
n = 11      T =     290510  C =      58786  A = 4.9418
n = 12      T =    1033410  C =     208012  A = 4.968
n = 13      T =    3707850  C =     742900  A = 4.991
n = 14      T =   13402695  C =    2674440  A = 5.0114
n = 15      T =   48760365  C =    9694845  A = 5.0295
n = 16      T =  178405155  C =   35357670  A = 5.0457
n = 17      T =  656043855  C =  129644790  A = 5.0603
n = 18      T = 2423307045  C =  477638700  A = 5.0735
```

It looks like $A(n)$ is stabilizing at around 5 . Another curious thing an astute observer might notice is how $T(n)$ and $C(n)$ seem to be related. We have

$$\begin{gathered} T(1)=C(2)=0+2=2 \\ T(2)=T(1)+C(3)=2+5=7 \\ T(3)=T(2)+C(4)=7+14=21 \\ T(4)=T(3)+C(5)=21+42=63 \\ T(5)=T(4)+C(6)=63+132=195 \\ \ldots \end{gathered}$$

Noticing this pattern, a reasonable conjecture would be the following:

$$T(n)=\sum_{i=2}^{n+1} C(i)$$

At this point, I have to confess I spent a few hours trying to prove this, with several approaches including induction but did not have much success. I plan to come back to this conjecture soon and see if I can find a proof, so I decided not to delay publishing this post due to a missing proof. I will update the article if/when I do find a proof. If you happen to think of a proof, do comment below and let me know!

Assuming the conjecture and what we know about the asymptotic behaviour of $\sum C_i$ from this paper by Sandro Mattarei, we have:

$$A(n) = \frac{T(n)}{C(n)} \sim \frac{\frac{4^{n+2}}{3(n+2)^{3/2}\sqrt{\pi}}}{\frac{4^n}{n^{3/2}\sqrt{\pi}}} \sim \frac{4^2 n^{3/2}}{3(n+2)^{3/2}} \sim \frac{4^2}{3} \sim 5.33$$

This matches our numerical estimate for $A(n)$. Of course, please do keep in mind all of this is assuming a conjecture that I have not yet proved.

Putting it all together, what does it all mean about the time and space complexity of our generation algorithm? It means we do, on average, a constant amount of work per balanced string generated by this algorithm, and hence the algorithm has *constant amortized time* or CAT. As for space complexity, it is trivially $O(n)$ which is as optimal as you can get--you have to store the string after all!

Can this algorithm be improved? A relatively simple look at the code seems to suggest we can make some improvements by noticing that we do not need to start from the very right side of the string each time. Since we replace the existing string with one of the form `)((...())...))`, immediately after that replacement we have better starting values for `i`, `o` and `c`. That is, we set `i` to the first known index of a `'('` and start there, and update `o` and `c` accordingly. This leads to the following optimized code:

```python
def gen_balanced_iterative_better(n):
    s = ['('] * n + [')'] * n
    starting_i, o, c = n + 1, 0, n
    while True:
        yield ''.join(s)
        for i in range(starting_i, 2 * n + 1):
            if i == 2 * n:
                return
            if s[-i] == '(':
                o += 1
                if c > o:
                    s[-i:] = [')'] + ['('] * o + [')'] * (c - 1)
                    starting_i, o, c = c, 0, c - 1
                    break
            else:
                c += 1
```

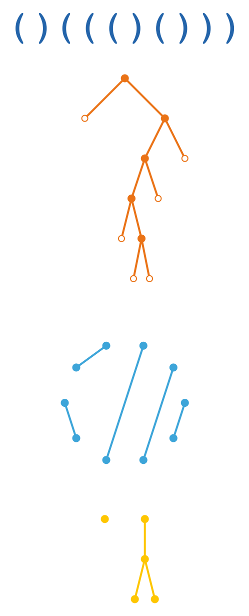
How much better of an algorithm is this? Let's calculate a similar table for $T'(n)$ for this algorithm instead. I'm skipping the code that generated this table since it's very similar to the code for the previous table.

```
n =  1      T' =         0 C =         1  A = 0.0
n =  2      T' =         3 C =         2  A = 1.5
n =  3      T' =        11 C =         5  A = 2.2
n =  4      T' =        34 C =        14  A = 2.4286
n =  5      T' =       104 C =        42  A = 2.4762
n =  6      T' =       326 C =       132  A = 2.4697
n =  7      T' =      1052 C =       429  A = 2.4522
n =  8      T' =      3483 C =      1430  A = 2.4357
n =  9      T' =     11777 C =      4862  A = 2.4223
n = 10      T' =     40507 C =     16796  A = 2.4117
n = 11      T' =    141283 C =     58786  A = 2.4033
n = 12      T' =    498521 C =    208012  A = 2.3966
n = 13      T' =   1776309 C =    742900  A = 2.391
n = 14      T' =   6382289 C =   2674440  A = 2.3864
n = 15      T' =  23097539 C =   9694845  A = 2.3825
```

Based on this benchmark, it looks like we are cutting the average number of comparisons by almost half. Not bad!

Binary Trees, Forests, Non-Crossing Pairs

Let's take a step back now and revisit the aforementioned equivalence of balanced parentheses and forests, binary trees and non-crossing pairs. Let's take a look at an example for `'()((()()))'`:



First, let's look at how the string `'()((()()))'` corresponds to a forest (shown in yellow). At the very top-level, there are two groups of parentheses: `'()'` and `'((()()))'`, each of which correspond to a tree. The first one, `'()'`, simply corresponds to a tree consisting of a single node. You can think of this as the base case. The second one, `'((()()))'`. has two leaves corresponding to the inner-most two instances of `'()'`, and ancestor nodes corresponding to the outer parentheses.

Generalizing the above observation, the simple way to define a forest corresponding to a balanced string is to define `'()'` to map to a single node, and define `'(' + x + ')'` to be a node with children recursively defined by `x`.

Let's turn this correspondence into working code. In the code, we define the nodes in the forest to be simply arrays, with each node simply containing its children (which means leaves will just be the empty array, `[]`). We will use a top-level array to contain all the top-level nodes of the forest.

For example, our string `'()((()()))'` will correspond to the following JavaScript array: `'[[], [[[], []]]]'`. Wait a second, you might be thinking. We barely did anything! The end result array looks almost exactly like the string we started with, we just added some commas in the right spots! That's very true. In fact, my first hacky string to forest code was the following:

```
function balancedStringToForest(s) {
    // This is a major hack to avoid writing parsing code :D
    var s2 = s
        .replace(/\(/g, '[')
        .replace(/\)/g, ']')
        .replace(/\]\[/g, '],[');
    return JSON.parse('[' + s2 + ']');
}
```

Here is what the non-hacky parsing code looks like, using a simple stack. It assumes a properly nested string, which, assuming our generation algorithm is bug-free, is a guarantee.

```
function balancedStringToForest(s) {
    var stack = [[]];
    for (var i = 0; i < s.length; i++) {
        if (s[i] == '(') {
            var node = [];
            stack[stack.length - 1].push(node);
            stack.push(node);
        } else {
            stack.pop();
        }
    }
    return stack[0];
}
```

What about non-crossing pairs, shown in light blue? To map a balanced string to a non-crossing pairs, we need to simply pair each index of `'('` in the string with the matching `')'`. This gives the following set of pairs: `(0, 1), (2, 9), (3, 8), (4, 5), (6, 7)`. Our parsing algorithm above can be modified to produce these matching pairs instead the forest:

```
function balancedStringToNonCrossingPairs(s) {
    var pairs = [];
    var stack = [];
    for (var i = 0; i < s.length; i++) {
        if (s[i] == '(') {
            stack.push(i);
        } else {
            pairs.push([stack.pop(), i]);
        }
    }
    return pairs;
}
```

And finally let's look at binary trees, shown in red. We will map a forest to a binary tree in the following way. First, insert a top-level node with all the top-level nodes as its children to turn the forest into a rooted tree (this is what our balanced string to forest code above did implicitly anyway by using a top-level array to store the top-level nodes of the forest). Now, starting with the root of the tree, create a binary tree by letting the left subtree of each node be the binary tree recursively obtained by mapping the node's children into a binary tree, and letting the right subtree be the binary tree obtained by recursively mapping all the node's right siblings to a binary tree. For example, given the forest `[[], [[[], []]]]`, we get the following binary tree: `[[],[[[[],[[],[]]],[]],[]]]`. This algorithm is likely easier described using plain code:

```
function forestToBinaryTree(forest) {
    if (forest.length > 0) {
        return [
            forestToBinaryTree(forest[0]),
            forestToBinaryTree(forest.slice(1))
        ];
    }
    return [];
}
```

Note: all I did here was show how the mappings work. I did not prove that they are in fact one-to-one and onto mappings. I am leaving it to the reader as an exercise to convince themselves that the mappings mentioned above are in fact bijections.

A Loopless Algorithm

Our best algorithm so far does a constant number of work on average (i.e., is CAT). However, the question remains: is it possible to do a constant amount of work per string generated even in the worst case? Another good question to ask is to go back to the problem of "jumpiness" that we observed earlier. Is it possible to generated all balanced strings in a way that minimizes the average number of character changes over all subsequent strings (or at least reduces the average relative to the lexicographic order that we are using)? The answer to both of these questions is yes, and there is a beautiful algorithm that achieves both simultaneously. The algorithm is due to Frank Ruskey and Aaron Williams and is detailed in the Paper "Generating Balanced Parentheses and Binary Trees by Prefix Shifts" which you can read here.

I won't go into detail of why the algorithm works but I will provide a brief description of it here and provide a Python implementation and then use our visualization method to visualize the generated sequence.

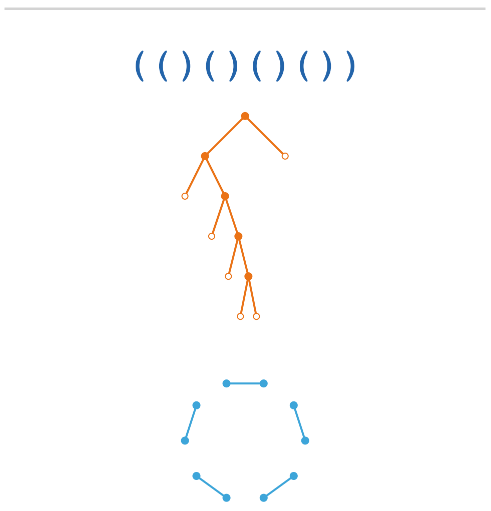
The algorithm's core idea is the "iterative successor rule" which is the following (with very minor changes): locate the leftmost 01 and suppose that its 1 is in position k . If the $(k+1)$ prefix shift is a balanced string then it is the successor; if it is not valid then the k prefix shift is the successor.

A k prefix shift here is changing $s_0, s_1, \ldots, s_{n-1}$ to $s_0, s_k, s_2, \ldots, s_{k-1}, s_{k+1}, \ldots, s_{n-1}$. A careful implementation of this "iterative successor rule" leads to the following code. I'm leaving the why and how to the paper as it does a great job of explaining it all. I'm mostly including the Python implementation here as a way of convincing you to read the paper as the code should look a bit like performing magic, especially if you have not seen a loopless combinatorial generation algorithm before.

```
def gen_balanced_rw(n):
    s = ['('] * n + [')'] * n
    x = n
    y = n
    yield ''.join(s)
    while x < 2 * n - 1:
        s[x - 1] = ')'
        s[y - 1] = '('
        x += 1
        y += 1
        if s[x - 1] == ')':
            if x == 2 * y - 2:
                x += 1
        else:
            s[x - 1] = '('
            s[1] = ')'
            x = 3
            y = 2

        yield ''.join(s)
```

Wait, "that has a loop!", you might be thinking. It does. But the critical difference here is that there is no nested loop. In other words, there is no loop per each output string. That's what makes this algorithm so beautiful: it is trivial to see that it does constant work for each string as it checks at most two conditionals per and performs a few extra operations per each string in the worst case, and only uses two variables to keep track of the state in addition to the generated string itself.

Here is the output of the algorithm visualized:



Sources And Further Reading

Knuth's Volume 4A of The Art of Computer Programming section 7.2.1.6 has an entire section on generating all trees with a wealth of detail for those wanting to learn more about the subject.

Ruskey and Williams's paper Generating Balanced Parentheses and Binary Trees by Prefix Shifts is the source of the loopless algorithm mentioned above.

Richard P. Stanley's Enumerative Combinatorics Volume 2 is an excellent source for learning more about Catalan numbers and the various combinatorial objects that they count.

Comments

José Barrios • 2 years ago

I'm happy to see you're back at it, great post!

2 | • Reply • Share ›

Recommended Articles

- Combinatorial Generation For Coding Interviews With Examples In Python
- Combinatorial Generation Using Coroutines With Examples in Python
- The Infinite In Haskell and Python
- From An Iterator of Iterators to Cantor's Paradise: A Deep Dive Into An Interview Question
- So How Do You Actually Calculate The Fibonacci Numbers?