A loopless and branchless $O(1)$ algorithm to generate the next Dyck word. *

Cassio Neri

19 July 2014

Abstract

Let `integer` be any C/C++ unsigned integer type up to 64-bits long. Given a Dyck word the following code returns the next Dyck word of the same size, provided it exists.

```
integer next_dyck_word(integer w) {
  integer const a = w & -w;
  integer const b = w + a;
  integer       c = w ^ b;
                c = (c / a >> 2) + 1;
                c = ((c * c - 1) & 0xaaaaaaaaaaaaaaaa) | b;
  return c;
}
```


1 Introduction

A $2n$-bits Dyck word is a string containing exactly n ones and n zeros and such that each of its prefix substrings contains no more zeros than ones.

Dyck words appear in a vast number of problems [1]. Consequently, generating them has many applications. For instance, if ones and zeros are replaced with openning and closing parentheses, then a Dyck word is a combination of n properly balanced pairs of parentheses. When one denotes a move rightwards and zero denotes a move upwards, a Dyck word represents a monotonic path along the edges of an $n \times n$ grid that starts at the lower left corner, finishes at the upper right corner and stays below diagonal. Figure 1 shows all 14 such paths on a 4×4 grid.

For the sake of clarity, another implementation of essentially the same algorithm is presented. This one works on strings of two different arbitrarily chosen symbols (e.g., opening and closing parentheses). Contrarily to the implementation in the abstract, the second one has explicit loops and branches and has $O(n)$ time complexity.

The apparent contradiction between being two implementations of "essentially the same" algorithm but one being loopless, brancheless and $O(1)$ and the other having loops, branches and being $O(n)$ can be explanained as follows.

To perform bitwise and arithmetic operations the hardware somehow "runs" loops on its transistors but, as far as I understand, for the sake of software complexity, these operations are considered as being loopless, branchless and $O(1)$. In that sense we can categorize the algorithm above as loopless, branchless and $O(1)$.

*Latest version and code available at github.com/cassioneri/Dyck

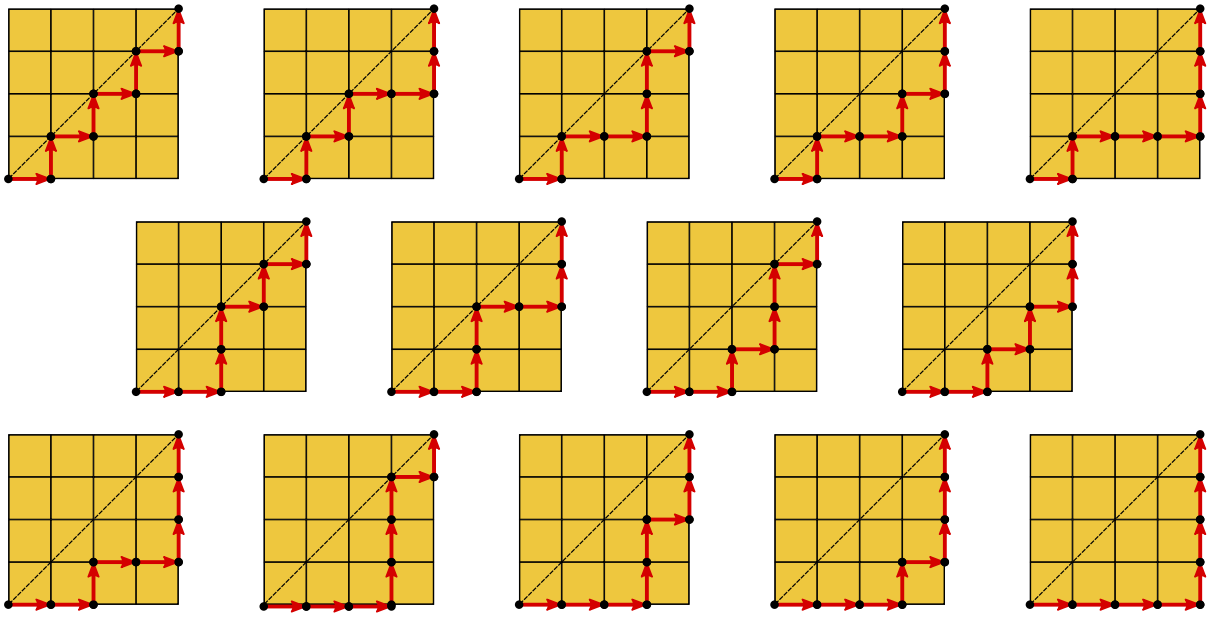

Figure 1: Monotonic paths on a 4×4 grid starting at the lower left corner, finishing at the upper right corner and staying below diagonal. There are 14 of them. The first path, for instance, corresponds to the Dyck word 10101010 and the fith one to 10111000.

The second implementation (through software code) simply makes explicit some loops and branches "ran" by the hardware when the first implementation is executed.

Both implementations are $O(1)$ in space.

Disclaimer: I am not a Computer Scientist and I am not aware of the state of the art. I do not claim the algorithm above has not been discovered/invented before but I believe there is a strict positive probability that it has not. I have performed a not very thoroughly search on the net and I have failed to find anything similar. The closest I know is the Gosper's hack [2] which, I must say, served as inspiration for the design of the algorithm above. Furthermore, the algorithm above borrows its first two lines from Gosper's hack.

2 Definitions

Let $n \in \mathbb{N}$ and $B = \{1, 0\}$. A $2n$-bits **word** is an element $w = (w_1, ..., w_{2n})$ of B^{2n}. Throughout n is fixed and, unless necessary, we omit the $2n$ from notation. Also, we most often drop parentheses and commas as well. (For instance, $(1, 0, 1, 0, 1, 0, 1, 0)$ becomes 10101010.) Finally, by abuse of notation, we identify $w = (w_1, ..., w_{2n})$ with the number whose binary expansion is w. More precisely, with

$$\sum_{i=1}^{2n} w_i 2^{2n-i}.$$

Thanks to this identification we can order words, talk about minimum, maximum, etc.

For $w \in B^{2n}$ and $i \in \{1, ..., 2n\}$ we define

$$N_1(w, i) := \#\{j \leq i \ ; \ w_j = 1\},$$
$$N_0(w, i) := \#\{j \leq i \ ; \ w_j = 0\}.$$

In plain English, $N_1(w,i)$ is the number of ones in w appearing before or at i-th position. Similarly, N_0 counts the number of zeros before or at position i.

We say that $w \in B^{2n}$ is a **Dyck word** if

$$N_1(w,i) \geq N_0(w,i) \quad \forall i \in \{1,...,2n\}, \tag{1}$$

with equality holding for $i = 2n$, that is,

$$N_1(w,2n) = N_1(w,2n) = n. \tag{2}$$

In plain English, before or at the i-th position, the number of ones must be no lesser than the number of zeros. In adition the total numbers of ones and zeros match.

For the mathematically trained eye, property (1) becomes easier to spot[1] when ones and zeros are replaced, respectively, with openning and closing parentheses. In this case, the Dyck word contains n pairs of correctly matched open and close parentheses. For instance, 10101010 becomes ()()()() and 10111000 becomes ()((())).

3 The minimum and the maximum Dyck word

The algorithm in the abstract generates the succesor of the given Dyck word. Therefore, to kick off and stop the generation of all words, we need to know the minimum and maximum Dyck word. (Recall that word size is fixed and "minimum" and "maximum" refer to the order of integer numbers.)

We claim that the minimum Dyck word is

$$\breve{w} := \underbrace{10\cdots10}_{n \text{ pairs}}$$

(or $()\cdots()$ in the parenthetical representation). More precisely, $\breve{w} = (\breve{w}_1, ..., \breve{w}_{2n}) \in B^{2n}$ is given by $\breve{w}_i = 1$, if i is odd, and $\breve{w}_i = 0$, if i is even.

Before proving our claim, we compute $\breve{w}$:

$$\begin{aligned}\breve{w} &= 2^{2n-1} + 2^{2n-3} + \cdots + 2^3 + 2^1 \\ &= 2\cdot 2^{2n-2} + 2\cdot 2^{2n-4} + \cdots + 2\cdot 2^2 + 2\cdot 2^0 \\ &= 2\cdot 4^{n-1} + 2\cdot 4^{n-2} + \cdots + 2\cdot 4 + 2\cdot 4^0 = 2\sum_{i=0}^{n-1} 4^i = \frac{2}{3}(4^n - 1).\end{aligned}$$

Calculating this number on real computers needs care to avoid overflow. For instance, if $n = 32$ then $4^n = 4^{32} = 2^{64}$ which is one more than the maximum natural number representable by a 64-bits unsigned integer type. In practice, however, this is a "minor" issue because generating all 64-bits Dyck words would take "forever" given that the number of $2n$-bits Dyck words grows factorially [1] with n.

We shall now prove that $\breve{w}$ is the minimum Dyck word. Suppose by contradiction that there's a Dyck word $w = (w_1, ..., w_{2n})$ such that $w < \breve{w}$. In particular, $w \neq \breve{w}$ and let i be the minimum index such that $w_i \neq \breve{w}_i$. Since $w < \breve{w}$, we have $w_i = 0$ and $\breve{w}_i = 1$. It follows that i is odd. By construction of $\breve{w}$, $N_1(\breve{w}, i-1) = N_0(\breve{w}, i-1)$ because $i-1$ is even. The same holds for w because it shares the first $i-1$ bits with $\breve{w}$. Now, $w_i = 0$ and thus $N_0(w,i) = N_0(w,i-1) + 1$ and $N_1(w,i) = N_1(w,i-1)$ which yields $N_0(w,i) = N_1(w,i) + 1$, contradicting (1).

[1] At least for small values of n or not so small if you are a Lisp programmer.

Its much easier to see that the maximum Dyck word is

$$\hat{w} := \underbrace{1\cdots1}_{n \text{ times}}\ \underbrace{0\cdots0}_{n \text{ times}},$$

which values $2^{2n} - 2^n$.

4 The next Dyck word

Let $w = (w_1, ..., w_{2n})$ be a Dyck word. In this section we characterize the Dyck word that succeeds w, that is, the smallest Dyck word which is greater than w.

Assuming that $w \neq \hat{w}$, *i.e.*, w is not the maximum Dyck word, there exists at least one index $i \in \{1, ..., 2n\}$ such that $w_i = 0$ and $w_{i+1} = 1$. Let k be the maximum of such indices. Since a Dyck word cannot start with a zero or finish with a one, we have $1 < k < 2n$. Then, after position k there is no bit zero followed by a bit one. More precisely, there's a (possiblye empty) sequence of ones followed by a sequence of zeros up to the end. Hence, w has this form:

$$w = (\underbrace{w_1, ..., w_{k-1}}_{\text{prefix}}, \underbrace{0}_{w_k}, \underbrace{1}_{w_{k+1}}, \underbrace{1, ..., 1}_{x \text{ times}}, \underbrace{0, ..., 0}_{y \text{ times}}),$$

with $x \geq 0$ and $y > 0$. We shall prove that the successor $\tilde{w}$ of w has this form:

$$\tilde{w} = (\underbrace{w_1, ..., w_{k-1}}_{\text{prefix}}, \underbrace{1}_{\tilde{w}_k}, \underbrace{0}_{\tilde{w}_{k+1}}, \underbrace{0, ..., 0}_{y-x \text{ times}}, \underbrace{1, 0, ..., 1, 0}_{x \text{ pairs of 1, 0}}).$$

Notice that the prefixes of w and $\tilde{w}$ are the same.

First we shal show that $\tilde{w}$ is well defined, that is, $y - x \geq 0$; the size of $\tilde{w}$ is $2n$; and $\tilde{w}$ verifies the properties (1) and (2).

The total number of ones in w is $N_1(w, k-1)+1+x$ and the total number of zeros is $N_0(w, k-1)+1+y$. Since w is a Dyck word, these numbers match and we obtain $N_1(w, k-1) + 1 + x = N_0(w, k-1) + 1 + y$. Hence,

$$N_1(w, k-1) + 1 = N_0(w, k-1) + 1 + y - x \Longrightarrow y - x = N_1(w, k-1) - N_0(w, k-1). \tag{3}$$

Again, because w is a Dyck word, $N_1(w, k-1) - N_0(w, k-1)$ is positive and so is $y - x$.

The last two segments of w have total size $x + y$ and the last two segments of $\tilde{w}$ have total size $y - x + 2x = x + y$. Hence $\tilde{w}$ is also $2n$-bits long and $k + 1 + y - x = 2(n - x)$.

Because of the common prefix with w, $\tilde{w}$ verifies property (1) for any $i < k-1$. Obviously, it also verifies (1) for $i = k$ ($\tilde{w}$ gets an extra 1).

Notice that for any $i \in \{k+1, ..., 2(n-x)\}$, $\tilde{w}_i = 0$. Hence, if $\tilde{w}$ fails to verify (1) for any i in this set of indices, then it fails to verify (1) for $i = 2(n-x)$. However, $N_1(\tilde{w}, 2(n-x)) = N_1(w, k-1) + 1$ and $N_0(\tilde{w}, 2(n-x)) = N_0(w, k-1)+1+y-x$ and from (3) we obtain that these numbers are equal. This proves that (1) holds for $i \leq 2(n-x)$. Furthermore, we have proven that $N_1(\tilde{w}, 2(n-x)) = N_0(\tilde{w}, 2(n-x))$.

From the $2(n-x)$-th bit onwards, the sequence is alternating and it follows that

$$\tilde{w}_i = 0 \quad \text{and} \quad N_1(\tilde{w}, i) = N_0(\tilde{w}, i), \quad \forall i \geq 2(n-x) \text{ that is even.} \tag{4}$$

To prove that $\tilde{w}$ is the smallest Dyck word which is greater than w, assume there's another Dyck word $v = (v_1, ..., v_{2n})$ such that $w < v < \tilde{w}$. Well, obviously, v must have the same prefix as w and $\tilde{w}$. What about v_k? It's either zero or one and we slipt in two cases.

If $v_k = 0 = w_k$ then, because $w < v$ and $w_i = 1$ for all $i \in \{k+1, ..., k+1+x\}$, the common part between v and w must span up to index $k+1+x$. For $i > k+1+x$, $w_i = 0$ and for $w < v$ to hold, at least one of the last y bits of v must be one. Then the number of ones in v is greater than the number of ones in w which violates the fact that any Dyck word has exactly n ones.

Similarly, if $v_k = 1 = \tilde{w}_k$, then because $v < \tilde{w}$ and $\tilde{w}_i = 0$ for $i \in \{k+1, ..., 2(n-x)\}$ the common part between v and $\tilde{w}$ must span up to index $2(n-x)$. But, since $v \neq \tilde{w}$, there exists $i > 2(n-x)$ such that $v_j \neq \tilde{w}_j$. Let i be the minimum of such indices. For $v < \tilde{w}$ to hold, it is necessary that $v_i = 0$ and $\tilde{w}_i = 1$. From (4) it follows that i is odd and $i - 1 \geq 2(n-x)$ is even. Again from (4) we obtain $N_1(\tilde{w}, i-1) = N_0(\tilde{w}, i-1)$ and the same holds for v thanks to its common part with $\tilde{w}$. But $v_i = 0$ and thus $N_1(v, i) = N_1(v, i-1)$ and $N_0(v, i) = N_0(1, i-1) + 1 = N_1(v, i-1) + 1$ which violates (1).

5 Two implementations

The following C++ function implements the algorithm presented in the previous section. It does not allocate memory for the output and, instead, performs the manipulations in-place. It makes no use of helper containers and, therefore, its $O(1)$ on space.

The algorithm scans the word backwards up to a certain point. Then it advances forward changing the bits up to the rightmost. Hence, in the worse case, the program scans the whole word twice implying complexity $O(n)$ on time.

The function takes a Dyck word `w` made of `ones` and `zeros` (provided as arguments) and transforms it into the next Dyck word, if it exists, otherwise, it clears the word. The behaviour is undefined if `w` is not a Dyck word of `ones` and `zeros`.

```
void next_dyck_word(std::string& w, char const c1, char const c0) {

  unsigned const m = w.size() - 1;
  unsigned       y = 0;
  unsigned       x = 0;

  for (unsigned i = m; i > 0; --i) {

    if (w[i] == c0)
      ++y; // Counter for trailing zeros.

    else if (w[i - 1] == c0) {

      // Found greatest i such that w[i] = c0 and w[i + 1] = c1.

      // Change these two chars.
      w[i - 1] = c1;
      w[  i  ] = c0;

      // Overwrite the following next y - x chars to c0.
      for (y = y - x; y != 0; --y)
        w[++i] = c0;

      // Overwrite the remaining chars with alternating ones and zeros.
      while (i < m) {
        w[++i] = c1;
```

```
        w[++i] = c0;
      }
      return;
    }

    else
      ++x; // Counter for ones that precede the trailing zeros.
  }
  w.clear(); // Failed to produce a Dyck word, then clear w.
}
```

We shall consider now the short implementation shown in the abstract. Similarly to the longer implementation, the short one assumes that the input is a Dyck word. If it is not, then this is a pre-condition violation which yields undefined behaviour. In addition (and opposite to the implementation above), another pre-condition is that the input is not the maximum Dyck word of its size. Failing to verify this condition, again, produces undefined behaviour.

For easy of reference (and because it is very short) we trascript the algorithm here.

```
integer next_dyck_word(integer w) {
  integer const a = w & -w;
  integer const b = w + a;
  integer       c = w ^ b;
                c = (c / a >> 2) + 1;
                c = ((c * c - 1) & 0xaaaaaaaaaaaaaaaa) | b;
  return c;
}
```

The abstract says that `integer` is a C/C++ *unsigned* integer type and the first line in the function's body takes the oposite of `w`! This is intended and works as expected on C and C++ conforming implementations because on these systems unsigned integer types have 2^N-modular arithmetics, where N is the size in bits of the integer type.

Other programming languages might have different integer types and, at this point, it is useful to list the properties the type must verify for the algorithm to work.

The first rule is obvious but worth saying: the type must implement the usual[2] binary representation of integer numbers. In particular, all values in the range $[0, 2^N - 1[$ are representable.

2^N-modular arithmetics is sufficient but not necessary and only the opposite of `w` must be as per 2^N-modular arithmetics (simply put the opposite of w is $2^N - w$). For the other operations, usual arithmetic rules are enough because the (mathematical) results of additions, subtractions, multiplications and divisions stay in the $[0, 2^N - 1[$ range. Moreover, `a` is a divisor of `c` (see below) and, therefore, no truncation or division by zero occurs.

We assume usual semantics also for the bitwise operators `&`, `^`, and `|`. Finally, the right shift rotation inserts zeros in the gaps on the left but if, this is not the case, we can replace the right shift of two bits by a division by four, provided that the division truncates the result.

The pleasure of verifying the details that the arithmetic and bitwise operations above reproduce the construction explained in previous section is left to the reader. I shall provide an overall picture though.

Let $\tilde{k}$, x, y and $\tilde{w}$ be as in the previous section.

The expression `w & -w`, assigned to `a`, yields a number whose bits are all zero but the one located

[2] For some definition of usual (e.g. two's complement).

at position $k+1+x$. The result of `w + a`, assigned to `b`, is a number that matches $\tilde{w}$ up to position $k+1+y-x=2(n-x)$ and have only zeros afterwards. Hence, the first two lines do almost everything. The following three fill up the last $2x$ positions with the sequence of ones and zeros.

The first value assigned to `c` is a number with all bits zero but $x+2$ of them which form a contiguous sequence finishing at position $k+1+x$. Recall that at this position lies the only non null bit of `a`. Hence, `c / a` has the effect of moving the sequence of $x+2$ ones in `c` all the way up to the right. Shifting 2 extra bits produces a number with all its x least significant bits set to one, that is, 2^x-1. After the addition to 1, `c` gets the value 2^x. Squaring produces 2^{2x} and decrementing yields $2^{2x}-1$ which is a mask for the $2x$ rightmost bits.

Now the magic number comes in (at this point, you have probably guessed what this is). It is the 64-bits number whose binary expansion is an alternating sequence of ones and zeros or, in other terms, it is the minimum 64-bits Dyck word.

Applying operator `&` to the mask and the magic number produces the $2x$-long sequence of ones and zeros needed to fill the gap in `b`. The operator `|` completes the task.

6 Acknowledgements

I thank my friend Prof. Lorenz Schneider who introduced me to the Gosper's hack about three years ago. I am also grateful towards *mosh111*, whoever he/she is, who introduced me to the problem of Dyck word generation through his/her post in [3] 3 days ago.

References

[1] Wikipedia, *Catalan numbers*, http://en.wikipedia.org/wiki/Catalan_number.

[2] Wikipedia, *Combinatorial number system*, http://en.wikipedia.org/wiki/Combinatorial_number_system.

[3] CareerCup, *Implement an algorithm to print all valid (e.g., properly opened and closed) combinations of n-pairs of parentheses*, http://www.careercup.com/question?id=4815516967370752.