

# Énumération de mots de Dyck de taille fixée, une version récursive en C

Michel Billaud (michel.billaud@laposte.net)

29 novembre 2021

## Table des matières

<b>1</b>	<b>Objectif</b>	<b>1</b>
1.1	Grammaire context-free des mots de Dyck . . . . .	2
1.2	Un algorithme non déterministe . . . . .	2
<b>2</b>	<b>Génération de tous les mots de taille <math>2n</math></b>	<b>2</b>
2.1	La fonction de génération . . . . .	3
2.1.1	Exemple d'utilisation . . . . .	3
2.1.2	Code de la fonction <code>gen()</code> . . . . .	3
2.2	Code de <code>gen1()</code> . . . . .	4
2.3	La fonction <code>gen2()</code> . . . . .	4
<b>3</b>	<b>Conclusion</b>	<b>5</b>
<b>4</b>	<b>Annexe, en Python avec des lambdas</b>	<b>5</b>



Ce texte fait partie d'une petite collection de notes mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 2.0 France.

- Les notes sont publiées dans <https://www.mbillaud.fr/notes/>
- Sources dans <https://github.com/MichelBillaud/notes-diverses>

Dernières corrections : 29 novembre 2021.

## 1 Objectif

Les *mots de Dyck* correspondent aux systèmes bien formés de parenthèses. On peut les voir comme des mots sur un alphabet à deux lettres  $A = \{a, b\}$

- qui ont autant de  $a$  que de  $b$ ,
- qui ont, quand on les coupe en 2, au moins autant de  $a$  à gauche que de  $b$ .

Exemple `aabbabaabb` est un mot de Dyck, de longueur 10.

Il existe des techniques bien connues pour énumérer ces mots :

- Ichiro Semba, Generation of all the balanced parenthesis strings in lexicographical order, Information Processing Letters, vol 12, num 4, pp 188-192 (1981)
- D. Knuth, The Art Of Computer Programming, vol 4 Combinatorial Algorithms, Part 1, 7.2.1.6, p 443 (2011).

Notre objectif est de montrer ici comment, dans un programme C, énumérer tous les mots de Dyck d'une taille fixée, **en nous basant sur la grammaire context-free** bien connue qui les engendre.

## 1.1 Grammaire context-free des mots de Dyck

Rappel les mots de Dyck sont générés par la grammaire context-free d'axiome  $S$  :

$$\begin{aligned} S &\rightarrow \epsilon \text{ (le mot vide)} \\ S &\rightarrow a.S.b.S \end{aligned}$$

Conséquence sur les mots d'une taille donnée : en remarquant que

- si  $u \in D_{2p}$ , et  $v \in D_{2q}$ , alors  $w = a.u.b.v \in D_{2n}$ , avec  $n = p + q + 1$
- et réciproquement, la grammaire n'étant pas ambiguë, la décomposition d'un mot de  $D_{2n}$  est unique.

Donc on peut obtenir tout mot de taille  $2n$  (non nulle) par une unique combinaison de 2 mots plus petits.

## 1.2 Un algorithme non déterministe

Pour générer au hasard un mot de Dyck de taille  $2n$ , l'entier  $n$  étant fixé on peut utiliser l'algorithme récursif non déterministe suivant :

- si  $n = 0$  le résultat est  $\epsilon$
- sinon
  - choisir un entier  $p$  entre 1 et  $n$ ,
  - générer un mot  $u$  de taille  $2p$ ,
  - générer un mot  $v$  de taille  $2q$ , avec  $q = n - p - 1$
  - le résultat est  $a.u.b.v$

Ce algorithme est susceptible de fournir tous les mots de taille  $2n$ , sans exception.

On peut aussi le voir comme un algorithme à backtracking, si on s'autorise à revenir en arrière dans les étapes pour trouver une autre solution.

## 2 Génération de tous les mots de taille $2n$

Notre but est d'obtenir des fonctions C qui permettent de traiter tous les mots de taille  $2n$ .

Ici nous essayons de coller au maximum à la définition récursive pour générer les mots.

## 2.1 La fonction de génération

La fonction que nous présentons combine *génération* et *traitement* de chacun des mots de Dyck d'une taille donnée.

### 2.1.1 Exemple d'utilisation

```
int main()
{
    printf("Enumeration des mots de Dyck de taille 6\n");
    int n = 0;
    gen(6, print_dyck_word_with_counter, &n);
}
```

Le premier paramètre indique la taille voulue, le second est une fonction à appliquer à chaque solution dans un certain contexte (3ième paramètre).

La fonction `print_dyck_word_with_counter` est un exemple d'utilisation, dans lequel le contexte est l'adresse d'un entier qui sert de compteur :

```
void print_dyck_word_with_counter(int n, char t[], void *context) {
    int *counter = context;
    *counter += 1;
    printf("%d : ", *counter);
    for (int i=0; i<n; i++) {
        printf("%c", t[i]);
    }
    printf("\n");
}
```

l'affichage produit est le suivant :

```
Enumeration des mots de Dyck de taille 6
1 : aaabbb
2 : aababb
3 : aabbab
4 : abaabb
5 : ababab
```

### 2.1.2 Code de la fonction gen()

```
typedef void (*ACTION)(int n, char t[], void *context);

void gen(int n, ACTION action, void *context) {
    int t[n];
    gen1(n, t, 0, n, action, context);
}
```

La fonction `gen` reçoit comme paramètres

- la taille choisie,
- l'action à effectuer sur les solutions (fonction à appliquer + son contexte).

Elle alloue un tableau `t`, et effectue un appel à `gen1` qui signifie

- dans le tableau `t` de taille `n`,
- remplir la tranche qui va de l'indice 0 à `n` (non compris) de toutes les façons possibles,
- et pour chaque possibilité, appliquer au tableau la fonction dans son contexte.

## 2.2 Code de `gen1()`

La fonction `gen1()` visite successivement toutes les manières de remplir la tranche d'indices `begin` à `end-1`, et leur applique une action.

- si la tranche est vide, l'action peut être appliquée au tableau
- si non, la fonction essaie différents “découpages” sous la forme “*a.u.b.v*” et se rappelle récursivement pour remplir la partie “*u*”, en indiquant comme continuation (`ctx`) qu'il faudra ensuite
  - remplir la partie “*v*”,
  - appliquer l'action au tableau quand il sera rempli.

```
void gen1(int n, char t[], int begin, int end, ACTION action, void *contexte)
{
    if (begin == end) {
        action(n, t, contexte);
    } else {
        t[begin] = 'a';
        for (int middle = end; middle > begin; middle -= 2) {
            t[middle-1] = 'b';
            struct ContinuationContext ctx = {
                .begin = middle,
                .end = end,
                .action = action,
                .context = contexte
            };
            gen1(n, t, begin+1, middle-1, &gen2, &ctx);
        }
    }
}
```

La boucle `for` est en ordre descendant pour fournir les solutions dans l'ordre lexicographique (en supposant que `a` précède `b`).

## 2.3 La fonction `gen2()`

La fonction `gen2` récupère les éléments du contexte d'origine, et rappelle `gen1` pour remplir la partie droite du mot :

```
struct ContinuationContext {
    int middle, end;
    ACTION action;
    void *context;
};

void gen2(int n, char t[], void *ctx)
```

```

{
    struct ContinuationContext *context = ctx;
    gen1(n, t, context->middle, context->end, context->action, context->context);
}

```

### 3 Conclusion

On effectue donc ici un parcours récursif de l'espace des solutions, avec application d'un traitement passé en paramètre.

En l'état actuel des choses (2021), le langage C ne propose pas de “lambdas”, ce traitement est décrit par transmission d'un pointeur de fonction et un pointeur vers un contexte.

### 4 Annexe, en Python avec des lambdas

```

def gen(size, process):
    buffer = ['x'] * size
    gen1(buffer, 0, size, process)

def gen1(buffer, begin, end, process) :
    if begin == end:
        process(buffer)
    else:
        buffer[begin] = 'a'
        for middle in range(end, begin, -2) :
            buffer[middle-1] = 'b'
            gen1(buffer, begin+1, middle-1,
                lambda b: gen1(buffer, middle, end, process))

```

```
gen(6, lambda l : print(l))
```

Exécution

```

['a', 'a', 'a', 'b', 'b', 'b']
['a', 'a', 'b', 'a', 'b', 'b']
['a', 'a', 'b', 'b', 'a', 'b']
['a', 'b', 'a', 'a', 'b', 'b']
['a', 'b', 'a', 'b', 'a', 'b']

```