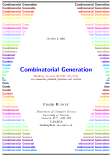
Comptage et énumération de structures de données: Algorithmes efficaces et implantations optimisées

Florent Hivert

Mél : `Florent.Hivert@lri.fr`
Adresse universelle : `http://www.lri.fr/~hivert`

Références



- Frank Ruskey, *Combinatorial Generation* `doi:10.1.1.93.5967`, 2003, non publié
- A. Nijenhuis and H.S. Wilf, *Combinatorial algorithms*, 2nd ed., Academic Press, 1978 `http://www.math.upenn.edu/~wilf/website/CombinatorialAlgorithms.pdf`
- The (Combinatorial) Object Server : `http://sue.csc.uvic.ca/~cos/`
- The On-Line Encyclopedia of Integer Sequences `http://oeis.org`

Algorithmes combinatoires

Manipulation d'ensembles finis, mais souvent très grand...

Algorithmes et implantations efficaces pour

- Compter, trouver la liste, itérer
- recherche d'un élément
- Tirage aléatoire équitable

Exemple d'ensemble finis...

Suite de n-bits :

0 1

00 01 10 11

000 001 010 011 100 101 110 111

0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111

Cardinal (nombre d'éléments) : `https://oeis.org/A000079`

$$2^n : 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 \ldots$$

Permutés de $[1, 2, \ldots, n]$

1

12 21

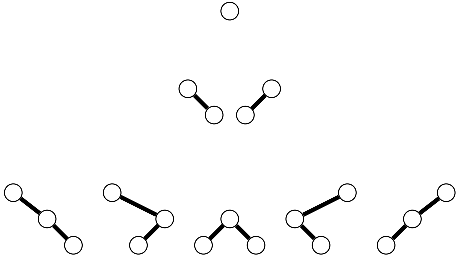
123 132 213 231 312 321

1234 1243 1324 1342 1423 1432 2134 2143 2314 2341 2413 2431
3124 3142 3214 3241 3412 3421 4123 4132 4213 4231 4312 4321

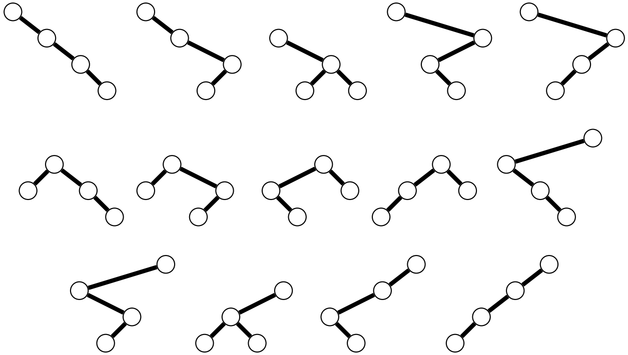
Cardinal (nombre d'éléments) : `https://oeis.org/A000142`

$n!$: $1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800 \ldots$

Arbres binaires à n noeuds



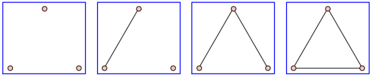
Arbres binaires à n noeuds

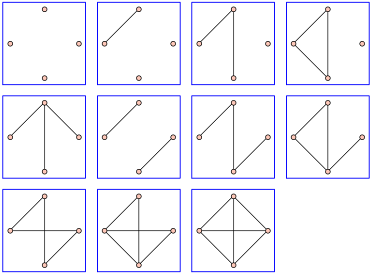


Cardinal (nombre d'éléments) : `https://oeis.org/A000142`

Cat(n) : $1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012 \ldots$

Les graphes à n noeuds :

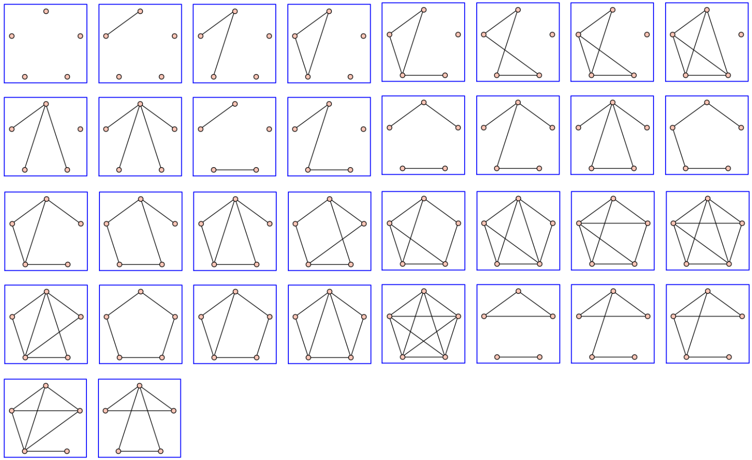




Cardinal (nombre d'éléments) : `https://oeis.org/A000088`

$\mathrm{Gr}(n)$: $1, 2, 4, 11, 34, 156, 1044, 12346, 274668, 12005168, 1018997864\ldots$

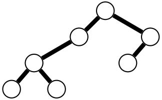
Les graphes à 5 noeuds :



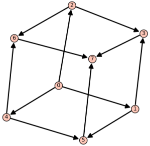
Manipulation d'ensembles finis :

... mais souvent très grand ...

- suites de 64 bits : `0xce24762189cdef0d`
- permutés d'un tableaux : `[5,3,6,4,1,2]`
- arbres binaires à 7 noeuds :



- graphes à 8-sommets :



- document XML à *n* balises
- programmes à *n* caractères en C, chemin d'execution

Algorithmes combinatoires

Soit S un ensemble **fini**.
On souhaite écrire les algorithmes suivants :

- count retourne le nombre d'éléments de S
- list retourne la liste des éléments de S
- iter itère sur les éléments de S
- unrank retourne le i-ème élement de la liste des éléments de S
- rank étant donné $s \in S$ retourne sa position dans la liste
- first retourne le premier élément de la liste
- next étant donné $s \in S$ retourne le suivant dans la liste
- random retourne un $s \in S$ au hasard de manière équitable

Applications

- recherche de solution par la force brute
- analyse d'algorithme, complexité
- tests de programme, de système
- recherche de failles, fuzzing
- bio-informaique, chimie, physique statistique

Algorithme list

Soit S l'ensemble des suites de 4 bits

Algorithme (list)

list(S) *retourne la liste des éléments de* S

Note : il faut fixer un ordre !

Par exemple pour l'ordre lexicographique :

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$
$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Algorithme list

Soit S l'ensemble des suites de 4 bits

Algorithme (list)

list(S) *retourne la liste des éléments de* S

Note : il faut fixer un ordre !

Par exemple pour l'ordre lexicographique :

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111

1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

Algorithme list

Soit S l'ensemble des suites de 4 bits

Algorithme (list)

list(S) *retourne la liste des éléments de* S

Note : il faut fixer un ordre !

Par exemple pour l'ordre lexicographique :

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$
$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Algorithme list

- Version récursive :

$$B_n = 0 \cdot B_{n-1} \;\cup\; 1 \cdot B_{n-1}$$

- Version Itérative en utilisant la base 2.

Algorithme list

- Version récursive :

$$B_n = 0 \cdot B_{n-1} \ \cup \ 1 \cdot B_{n-1}$$

- Version Itérative en utilisant la base 2.

Algorithme count

Algorithme (count)

count(S) *retourne le nombre d'éléments de S (la cardinalité de S).*

Soit S l'ensemble des suites de 4 bits

$$\text{count}(S) = 16$$

Algorithme count

Algorithme (count)

count(S) *retourne le nombre d'éléments de S (la cardinalité de S).*

Soit S l'ensemble des suites de 4 bits

$$\text{count}(S) = 16$$

Algorithme count

Algorithme (count)

count(S) *retourne le nombre d'éléments de S (la cardinalité de S).*

Soit S l'ensemble des suites de 4 bits

$$\mathrm{count}(S) = 16$$

Algorithme unrank

Algorithme (unrank)

unrank(S, i) *retourne le i-ème élement de la liste des éléments de S pour* $0 \leq i < \mathrm{count}(S)$.

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\mathrm{unrank}(S, 11) = 1011$$

Algorithme unrank

Algorithme (unrank)

unrank(S, i) *retourne le i-ème élement de la liste des éléments de* S *pour* $0 \leq i < \text{count}(S)$.

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\text{unrank}(S, 11) = 1011$$

Algorithme rank

Algorithme (rank)

rank(S, s) *retourne la position de* $s \in S$ *dans la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\mathrm{rank}(S, 1011) = 11$$

Algorithme rank

Algorithme (rank)

rank(S, s) *retourne la position de* $s \in S$ *dans la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\mathrm{rank}(S, 1011) = 11$$

Algorithme first

Algorithme (first)

first(S) *retourne le premier élément de la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\mathrm{first}(S) = 0000$$

Algorithme first

Algorithme (first)

first(S) *retourne le premier élément de la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\mathrm{first}(S) = 0000$$

Algorithme next

Algorithme (next)

next(S, s) *retourne l'élément qui suit $s \in S$ dans la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111

1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

Alors :

$$\mathrm{next}(S, 1011) = 1100$$

et

$$\mathrm{next}(S, 1111) = \text{Erreur ou Exception}$$

Algorithme next

Algorithme (next)

next(S, s) *retourne l'élément qui suit $s \in S$ dans la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\text{next}(S, 1011) = 1100$$

et

$$\text{next}(S, 1111) = \text{Erreur ou Exception}$$

Algorithme next

Algorithme (next)

next(S, s) *retourne l'élément qui suit* $s \in S$ *dans la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\text{next}(S, 1011) = 1100$$

et

$$\text{next}(S, 1111) = \text{Erreur ou Exception}$$

Algorithme next

Algorithme (next)

next(S, s) *retourne l'élément qui suit* $s \in S$ *dans la liste*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\text{next}(S, 1011) = 1100$$

et

$$\text{next}(S, 1111) = \text{Erreur ou Exception}$$

Algorithme random

Algorithme (random)

random(S, s) *retourne un element de s au hasard de manière équitable*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

random(S) peut retourner 0011

Algorithme random

Algorithme (random)

random(S, s) *retourne un element de s au hasard de manière équitable*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

random(S) peut retourner 0011

Algorithme random

Algorithme (random)

random(S, s) *retourne un element de s au hasard de manière équitable*

Soit S l'ensemble des suites de 4 bits dans l'ordre lexicographique

$$0000,\ 0001,\ 0010,\ 0011,\ 0100,\ 0101,\ 0110,\ 0111$$

$$1000,\ 1001,\ 1010,\ 1011,\ 1100,\ 1101,\ 1110,\ 1111$$

Alors :

$$\text{random}(S) \text{ peut retourner } 0011$$

Algorithme iter

Algorithme (iter)

iter(S) *permet d'itérer sur les éléments de S*

Différents protocoles :

- objet avec une méthode `next` et exception (Python)
- objet avec méthodes `next` et `hasNext` (Java)
- objet avec passage au suivant ++, déréférencement $*$ et garde de fin (C++)
- fonction de rappel (callback)
- modèle producteur-consommateur (par ex. threads)

Algorithme iter

Algorithme (iter)

iter(S) *permet d'itérer sur les éléments de S*

Différents protocoles :

- objet avec une méthode `next` et exception (Python)
- objet avec méthodes `next` et `hasNext` (Java)
- objet avec passage au suivant ++, déréférencement $*$ et garde de fin (C++)
- fonction de rappel (callback)
- modèle producteur-consommateur (par ex. threads)

Algorithme iter

Retenir (iter vs. list)

Intérêts de iter *par rapport à* list *:*

- *liste trop grande pour tenir en mémoire*
- *algorithme en place, meilleur utilisation des caches de la mémoire*
- *peut avoir une* ***complexité plus faible***

Algorithme iter

Retenir (iter vs. list)

Intérêts de iter *par rapport à* list *:*

- *liste trop grande pour tenir en mémoire*
- *algorithme en place, meilleur utilisation des caches de la mémoire*
- *peut avoir une* ***complexité plus faible***

Algorithme iter

Retenir (iter vs. list)

Intérêts de iter *par rapport à* list *:*

- *liste trop grande pour tenir en mémoire*
- *algorithme en place, meilleur utilisation des caches de la mémoire*
- *peut avoir une* ***complexité plus faible***

Algorithme iter

Retenir (iter vs. list)

Intérêts de iter *par rapport à* list *:*

- *liste trop grande pour tenir en mémoire*
- *algorithme en place, meilleur utilisation des caches de la mémoire*
- *peut avoir une* ***complexité plus faible***

Notion de classe combinatoire

Définition (Classe combinatoire)

On appelle ***classe combinatoire*** *un ensemble C dont les éléments e ont une taille (nommée aussi degrée) noté $|e|$ et tels que l'ensemble C_n des éléments de taille n est fini :*

$$\mathrm{count}(\{e \in C \mid |e| = n\}) < \infty$$

Complexité de list

Problème : liste des éléments de taille n.

Proposition

La complexité de list *ne peut être meilleure que* $O(n\,\mathrm{count}(C_n))$.

Complexité de list

Problème : liste des éléments de taille n.

Proposition

La complexité de list *ne peut être meilleure que* $O(n\,\mathrm{count}(C_n))$.

Complexité de iter

En revanche pour iter on peut obtenir

Définition

On dit qu'un algorithme est de complexité CAT (Constant Amortized Time) ***temps constant amortis*** *si en moyenne chaque appel prend un temps constant.*

Ici, le nombre d'appel à la méthode `next` de l'itérateur est $\text{count}(C_n)$. Il faut donc que

$$\frac{\text{Coût total des appels à } \texttt{next}}{\text{count}(C_n)} \in O(1)$$

Note : il n'y a pas de borne au coût d'un appel à la méthode `next`.

Complexité de iter

En revanche pour iter on peut obtenir

Définition

On dit qu'un algorithme est de complexité CAT (Constant Amortized Time) ***temps constant amortis*** *si en moyenne chaque appel prend un temps constant.*

Ici, le nombre d'appel à la méthode `next` de l'itérateur est $\mathrm{count}(C_n)$. Il faut donc que

$$\frac{\text{Coût total des appels à } \texttt{next}}{\mathrm{count}(C_n)} \in O(1)$$

Note : il n'y a pas de borne au coût d'un appel à la méthode `next`.

Complexité de iter

En revanche pour iter on peut obtenir

Définition

On dit qu'un algorithme est de complexité CAT (Constant Amortized Time) ***temps constant amortis*** *si en moyenne chaque appel prend un temps constant.*

Ici, le nombre d'appel à la méthode `next` de l'itérateur est $\mathrm{count}(C_n)$. Il faut donc que

$$\frac{\text{Coût total des appels à } \texttt{next}}{\mathrm{count}(C_n)} \in O(1)$$

Note : il n'y a pas de borne au coût d'un appel à la méthode `next`.