Algorithmes combinatoires : comptage, génération et tirage aléatoire

Florent Hivert

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

Références

- A. Nijenhuis and H.S. Wilf, *Combinatorial algorithms*, 2nd ed., Academic Press, 1978
  `http://www.math.upenn.edu/~wilf/website/CombinatorialAlgorithms.pdf`

- Frank Ruskey, *Combinatorial Generation*
  `doi:10.1.1.93.5967`, non publié

- 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 :

Exemples d'ensembles finis :

- suites de 64 bits
- permutés d'un tableaux
- arbres binaires à n-feuilles
- graphes à n-sommets
- arbres couvrants d'un graphe donné
- programmes à n caractères en C
- document XML à n balises

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

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

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)

$\text{next}(S, s)$ *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

$$\text{next}(S, 0000) = \text{Erreur ou Exception}$$

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

$$\text{next}(S, 0000) = \text{Erreur ou Exception}$$

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

$$\text{next}(S, 0000) = \text{Erreur ou Exception}$$

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

$$\text{next}(S, 0000) = \text{Erreur ou Exception}$$

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 :

random(S) 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 $\mathrm{count}(C_n)$. 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`.

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`.