Grammaires de descriptions d'objets

Florent Hivert

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

Objectifs : algorithmes génériques

- **Identifier les composants de base :**

  $\Longrightarrow$ Singleton, union, produit cartésien, ensemble et multiensemble. . .

- Comprendre comment composer les briques de base

  $\Longrightarrow$ grammaire de description, classe combinatoire

- $\Longrightarrow$ Algorithmes génériques

Objectifs : algorithmes génériques

- **Identifier les composants de base :**

  $\Longrightarrow$ Singleton, union, produit cartésien, ensemble et multiensemble...

- Comprendre comment composer les briques de base

  $\Longrightarrow$ grammaire de description, classe combinatoire

- $\Longrightarrow$ Algorithmes génériques

Union disjointe

Definition

On écrit $C = A \sqcup B$ et on dit que C est l'**union disjointe** de A et B si $C = A \sqcup B$ et $A \cap B = \emptyset$.

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) + \mathrm{count}(B)$
- On peut prendre : $\mathrm{list}(C) = \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$

Union disjointe

Definition

On écrit $C = A \sqcup B$ et on dit que C est l'**union disjointe** de A et B si $C = A \sqcup B$ et $A \cap B = \emptyset$.

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) + \mathrm{count}(B)$
- On peut prendre : $\mathrm{list}(C) = \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$

Itération sur une union disjointe

On fixe l'ordre d'énumération tel que

$$\mathrm{list}(A \sqcup B) := \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$$

Itération en Python :

```
def iterunion(A, B):
    for a in A:
        yield a
    for b in B:
        yield b
```

first, next sur une union disjointe

$$\mathrm{list}(A \sqcup B) := \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$$

```
def first_union(A, B):
    return A.first()

def next_union(A, B, x):
    if x in A:
        try:
            return A.next(x)
        except StopIteration:
            return B.first()
    else:
        return B.next(x)
```

rank sur une union disjointe

$$\text{list}(A \sqcup B) := \text{concat}(\text{list}(A), \text{list}(B))$$

```
def rank_union(A, B, x):
    if x in A:
        return A.rank(x)
    else:
        return A.count() + B.rank(x)

def unrank_union(A, B, i):
    if i < A.count():
        return A.unrank(i)
    else:
        return B.unrank(i - A.count())
```

Le principe de l'idée récursive

**Quand on a un'bonne idée,
on l'appliqu'récursivement :
on obtient le plus souvent
une bien meilleure idée !**

- Unions disjointes récursives

Le principe de l'idée récursive

Quand on a un'bonne idée,
on l'appliqu'récursivement :
on obtient le plus souvent
une bien meilleure idée !

- Unions disjointes récursives

Les chaînes de n-bits ayant k-bits à 1

Une chaîne de bit non vide commence soit par un 0, soit par un 1 :

$\mathrm{BitString}(n, k) = 0 \cdot \mathrm{BitString}(n-1, k) \sqcup 1 \cdot \mathrm{BitString}(n-1, k-1)$

Idem triangle de pascal :

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- $\mathrm{BitString}(n, k).\mathrm{count}() = \binom{n}{k}$

rank, unrank pour les chaînes de n-bits ayant k-bits à 1

```
def rank_BSnk(x):
    if not x:          # liste vide
        return 0
    if x[0] == 0:
        return rank_BSnk(x[1:])
    else:
        return binom(len(x)-1, sum(x)-1) + rank_BSnk(x[1:])

def unrank_BSnk(n, k, i):
    if n == 0:
        return []
    bn1k = binom(n-1, k)
    if i < bn1k:
        return [0]+unrank_BSnk(n-1, k, i)
    else:
        return [1]+unrank_BSnk(n-1, k-1, i-bn1k)
```

Le problème du calcul de la cardinalité

Problème

Le calcul récursif des coefficients binomiaux $\binom{n}{k}$ n'est pas efficace car on recalcule plusieurs fois la même chose.

Plus généralement, le calcul récursif des cardinalités sera très inefficace pour la même raison.

Parenthèse : mémoization et programmation dynamique

Retenir

- ***Mémoisation :*** *on mémorise tous les calculs pendant la récursion au momment où on les fait*
- ***Programmation Dynamique :*** *résoud les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.*

En général, la programmation dynamique est plus efficace mais plus longue à mettre en oeuvre : il faut avoir planifié l'utilisation de la mémoire.

Parenthèse : mémoization et programmation dynamique

Retenir

- ***Mémoisation :*** *on mémorise tous les calculs pendant la récursion au momment où on les fait*
- ***Programmation Dynamique :*** *résoud les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.*

En général, la programmation dynamique est plus efficace mais plus longue à mettre en oeuvre : il faut avoir planifié l'utilisation de la mémoire.

Parenthèse : mémoization et programmation dynamique

Retenir

- ***Mémoisation :*** *on mémorise tous les calculs pendant la récursion au momment où on les fait*
- ***Programmation Dynamique :*** *résoud les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.*

En général, la programmation dynamique est plus efficace mais plus longue à mettre en oeuvre : il faut avoir planifié l'utilisation de la mémoire.

Autre exemple : les permutations

Les permutés d'un ensemble $X := \{x_1, x_2, \ldots, x_n\}$:

$$\mathrm{Perm}\{1,2,3\} = 1 \cdot \mathrm{Perm}\{2,3\} \ \sqcup \ 2 \cdot \mathrm{Perm}\{1,3\} \ \sqcup \ 3 \cdot \mathrm{Perm}\{1,2\}$$

Plus généralement :

Retenir

Énumération lexicographique des permutations :

$$\mathrm{Perm}(X) = \bigsqcup_{i=1}^{n} x_i \cdot \mathrm{Perm}(X/\{x_i\})$$

- $\mathrm{Perm}(X).\mathrm{count}() = |X|!$

Généralisation : permuté d'un multiensemble

$\mathrm{Perm}\{1,1,2,3\} = 1\cdot\mathrm{Perm}\{1,2\} \sqcup 2\cdot\mathrm{Perm}\{1,1,3\} \sqcup 3\cdot\mathrm{Perm}\{1,1,2\}$

Notation : $\{1,1,2,3\} = 1^2 2^1 3^1$

$\mathrm{Perm}(1^2 2^3 3^1) = 1\cdot\mathrm{Perm}(1^1 2^3 3^1) \sqcup 2\cdot\mathrm{Perm}(1^1 2^2 3^1) \sqcup 3\cdot\mathrm{Perm}(1^1 2^3)$

Retenir

Énumération lexicographique des multi-permutations :

$$\mathrm{Perm}(X) = \bigsqcup_{i=1}^{n} x_i \cdot \mathrm{Perm}(X/\{x_i\})$$

Coefficient multinomiaux :

$$\binom{|I|}{i_1, i_2, \ldots, i_k} = \binom{|I|-1}{i_1-1, i_2, \ldots, i_k} + \binom{|I|-1}{i_1, i_2-1, \ldots, i_k} + \cdots + \binom{|I|-1}{i_1, i_2, \ldots, i_k-1}$$

$$\mathrm{Perm}(x_1^{i_1} \ldots x_k^{i_k}).\mathrm{count}() = \frac{(i_1+i_2+\cdots+i_k)!}{i_1!i_2!\ldots i_k!} = \binom{|I|}{i_1, i_2, \ldots, i_k}$$

Coefficient multinomiaux :

$$\binom{|I|}{i_1, i_2, \ldots, i_k} = \binom{|I|-1}{i_1 - 1, i_2, \ldots, i_k} + \binom{|I|-1}{i_1, i_2 - 1, \ldots, i_k} + \cdots + \binom{|I|-1}{i_1, i_2, \ldots, i_k - 1}$$

$$\mathrm{Perm}(x_1^{i_1} \ldots x_k^{i_k}).\mathrm{count}() = \frac{(i_1 + i_2 + \cdots + i_k)!}{i_1! i_2! \ldots i_k!} = \binom{|I|}{i_1, i_2, \ldots, i_k}$$

Le produit cartesien

Definition

On appelle **produit cartesien** de A et B l'ensemble C noté $C := A \times B$ défini par

$$C := \{(a, b) \mid a \in A, b \in B)\} .$$

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) \cdot \mathrm{count}(B)$
- On peut prendre la liste dans l'ordre lexicographique : $\mathrm{list}(C) = [(a_1, b_1), (a_1, b_2), (a_1, b_3), \dots (a_2, b_1), (a_2, b_2) \dots]$.

Le produit cartesien

Definition

On appelle **produit cartesien** de A et B l'ensemble C noté $C := A \times B$ défini par

$$C := \{(a, b) \mid a \in A, b \in B)\}.$$

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) \cdot \mathrm{count}(B)$
- On peut prendre la liste dans l'ordre lexicographique : $\mathrm{list}(C) = [(a_1, b_1), (a_1, b_2), (a_1, b_3), \ldots (a_2, b_1), (a_2, b_2) \ldots]$.

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 :*

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

Le produit cartesien gradué