

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 :

- ⇒ Singleton, union, produit cartésien, ensemble et multiensemble. . .

- Comprendre comment composer les briques de base

- ⇒ grammaire de description, classe combinatoire

- ⇒ Algorithmes génériques

Objectifs : algorithmes génériques

- Identifier les composants de base :

⇒ Singleton, union, produit cartésien, ensemble et multiensemble. . .

- Comprendre comment composer les briques de base

⇒ grammaire de description, classe combinatoire

- ⇒ 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 :

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

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 :

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

Itération sur une union disjointe

On fixe l'ordre d'énumération tel que

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

Itération en Python :

```
1      def iterunion(A, B):
2          for a in A:
3              yield a
4          for b in B:
5              yield b
```

first, next sur une union disjointe

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

```
1      def first_union(A, B):
2          return A.first()
3
4      def next_union(A, B, x):
5          if x in A:
6              try:
7                  return A.next(x)
8              except StopIteration:
9                  return B.first()
10         else:
11             return B.next(x)
```

rank sur une union disjointe

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

```
1      def rank_union(A, B, x):
2          if x in A:
3              return A.rank(x)
4          else:
5              return A.count() + B.rank(x)
6
7      def unrank_union(A, B, i):
8          if i < A.count():
9              return A.unrank(i)
10         else:
11             return B.unrank(i - A.count())
```


Le principe de l'idée réursive

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 :

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

Idem triangle de pascal :

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

■ $\text{BitString}(n, k). \text{count}() = \binom{n}{k}$

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

```
1  def rank_BSnk(x):
2      if not x:          # liste vide
3          return 0
4      if x[0] == 0:
5          return rank_BSnk(x[1:])
6      else:
7          return binom(len(x)-1, sum(x)-1) + rank_BSnk(x[1:])
8
9  def unrank_BSnk(n, k, i):
10     if n == 0:
11         return []
12     bn1k = binom(n-1, k)
13     if i < bn1k:
14         return [0]+unrank_BSnk(n-1, k, i)
15     else:
16         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 moment 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 moment 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 moment 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 $X := \{x_1, x_2, \dots, x_n\}$:

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

Plus généralement :

Retenir

Énumération lexicographique des permutations :

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

■ $\text{Perm}(X).\text{count}() = |X|!$

Généralisation : permuté d'un multiensemble

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

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

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

Retenir

Énumération lexicographique des multi-permutations :

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

Coefficient multinomiaux :

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

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

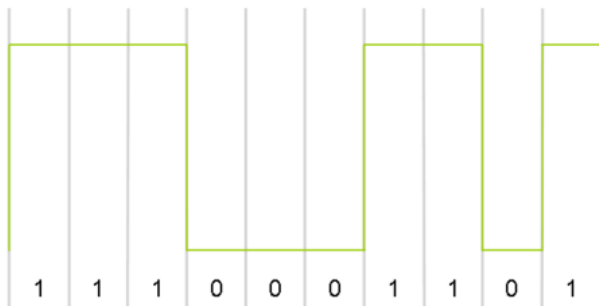
Coefficient multinomiaux :

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

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

Autre application : transmission en codage NRZ

Non Return to Zero, Manière très élémentaire pour transmettre de l'information sur un ligne : 0 : $-V$, 1 : $+V$



Source : https://fr.wikipedia.org/wiki/Non_Return_to_Zero

Perte de synchronisation en codage NRZ

S'il on envoie une suite trop longue de bits identiques, on perd la synchronisation.

Definition

Une séquence de longueur n est dite non-répétitive d'ordre k (abréviation $NR(k, n)$) si elle ne contient pas de séquence de plus de k bits identiques consécutifs.

Notation : $NR(k, n, c, i)$ les suites qui commencent par au plus c i :

- $NR(k, n, 0, 0) = 1 \cdot NR(k, n-1, k-1, 1)$
- $NR(k, n, c, 0) = 0 \cdot NR(k, n-1, c-1, 0) \sqcup 1 \cdot NR(k, n-1, k-1, 1)$
- Idem en échangeant les rôles de 0 et 1.

C'est un automate fini !

Perte de synchronisation en codage NRZ

S'il on envoie une suite trop longue de bits identiques, on perd la synchronisation.

Definition

Une séquence de longueur n est dite non-répétitive d'ordre k (abréviation $NR(k, n)$) si elle ne contient pas de séquence de plus de k bits identiques consécutifs.

Notation : $NR(k, n, c, i)$ les suites qui commencent par au plus c i :

- $NR(k, n, 0, 0) = 1 \cdot NR(k, n-1, k-1, 1)$
- $NR(k, n, c, 0) = 0 \cdot NR(k, n-1, c-1, 0) \sqcup 1 \cdot NR(k, n-1, k-1, 1)$
- Idem en échangeant les rôles de 0 et 1.

C'est un automate fini !

Union récursive et automates finis

Retenir

La méthode précédente fonctionne pour toute les automates finis déterministes.

$$\text{Lang}_0(\text{Term}) = \{\epsilon\}$$

$$\text{Lang}_n(E) = \bigsqcup_{E \rightarrow_a E'} a \cdot \text{Lang}_{n-1}(E')$$

Le produit cartésien

Definition

On appelle **produit cartésien** 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 :

- $\text{count}(C) = \text{count}(A) \cdot \text{count}(B)$
- On peut prendre la liste dans l'ordre lexicographique :

$$\begin{aligned} \text{list}(C) = & [(a_0, b_0), (a_0, b_1), (a_0, b_2), \dots, (a_0, b_l), \\ & (a_1, b_0), (a_1, b_1), (a_1, b_2), \dots, (a_1, b_l), \\ & (a_2, b_0), (a_2, b_1), (a_2, b_2), \dots, (a_2, b_l), \\ & \dots \end{aligned}] .$$

Le produit cartésien

Definition

On appelle **produit cartésien** 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 :

- $\text{count}(C) = \text{count}(A) \cdot \text{count}(B)$
- On peut prendre la liste dans l'ordre lexicographique :

$$\begin{aligned} \text{list}(C) = & [(a_0, b_0), (a_0, b_1), (a_0, b_2), \dots, (a_0, b_l), \\ & (a_1, b_0), (a_1, b_1), (a_1, b_2), \dots, (a_1, b_l), \\ & (a_2, b_0), (a_2, b_1), (a_2, b_2), \dots, (a_2, b_l), \\ & \dots \end{aligned}] .$$

Itération sur un produit cartésien

Ordre lexicographique :

Itération en Python :

```
1      def iter_cartprod(A, B):  
2          for a in A:  
3              for b in B:  
4                  yield (a, b)
```

first, next sur un produit cartésien

Ordre lexicographique :

```
1      def first_cartprod(A, B):
2          return (A.first(), B.first())
3
4      def next_cartprod(A, B, x):
5          (a , b) = x          # pattern matching
6          try:
7              return (a, B.next(b))
8          except StopIteration:
9              return (A.next(a), B.first())
```

rank sur un produit cartésien

Ordre lexicographique :

```
1      def rank_cartprod(A, B, x):
2          (a , b) = x          # pattern matching
3          A.rank(a)*B.count() + B.rank(b)
4
5      def unrank_cartprod(A, B, i):
6          c = B.count()
7          return (A.unrank(i // c), B.unrank(i % c))
```

Notion de classe combinatoire

Définition (Classe combinatoire)

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

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

Exemple :

- Les mots sur un alphabet où la taille est la longueur
- Les permutations de $\{1, \dots, n\}$ (taille = n)
- Les arbres binaires où la taille est le nombre de noeuds

L'union disjointe graduée

Si $\mathcal{C} = \mathcal{A} \sqcup \mathcal{B}$, les éléments de \mathcal{A} et \mathcal{B} gardent leur taille dans l'union disjointe graduée :

$$\mathcal{C}_n := \mathcal{A}_n \sqcup \mathcal{B}_n$$

Alors :

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

\implies On peut réutiliser tout ce que l'on a vu sur les unions disjointes.

L'union disjointe graduée

Si $\mathcal{C} = \mathcal{A} \sqcup \mathcal{B}$, les éléments de \mathcal{A} et \mathcal{B} gardent leur taille dans l'union disjointe graduée :

$$\mathcal{C}_n := \mathcal{A}_n \sqcup \mathcal{B}_n$$

Alors :

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

\implies On peut réutiliser tout ce que l'on a vu sur les unions disjointes.

Le produit cartésien gradué

Idée : les tailles (complexité, coût, nbr d'emplacements mémoires) s'ajoutent.

Definition

La taille de la paire $(a, b) \in \mathcal{A} \times \mathcal{B}$ est la somme des tailles :

$$|(a, b)|_{\mathcal{A} \times \mathcal{B}} := |a|_{\mathcal{A}} + |b|_{\mathcal{B}}$$

Le produit cartésien gradué

Retenir

Si $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ alors

$$\mathcal{C}_n = \bigsqcup_{i+j=n} \mathcal{A}_i \times \mathcal{B}_j$$

Calcul de la cardinalité :

$$|\mathcal{C}_n| = \sum_{i+j=n} |\mathcal{A}_i| \times |\mathcal{B}_j| = \sum_{i=0}^n |\mathcal{A}_i| \times |\mathcal{B}_{n-i}|$$

On peut alors prendre l'ordre union/lexicographique suivant :

$\mathcal{A}_0 \times \mathcal{B}_n$	$\mathcal{A}_1 \times \mathcal{B}_{n-1}$	$\mathcal{A}_2 \times \mathcal{B}_{n-2}$	\dots	$\mathcal{A}_n \times \mathcal{B}_0$
--------------------------------------	--	--	---------	--------------------------------------

Le produit cartésien gradué

Retenir

Si $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ alors

$$\mathcal{C}_n = \bigsqcup_{i+j=n} \mathcal{A}_i \times \mathcal{B}_j$$

Calcul de la cardinalité :

$$|\mathcal{C}_n| = \sum_{i+j=n} |\mathcal{A}_i| \times |\mathcal{B}_j| = \sum_{i=0}^n |\mathcal{A}_i| \times |\mathcal{B}_{n-i}|$$

On peut alors prendre l'ordre union/lexicographique suivant :

$\mathcal{A}_0 \times \mathcal{B}_n$	$\mathcal{A}_1 \times \mathcal{B}_{n-1}$	$\mathcal{A}_2 \times \mathcal{B}_{n-2}$	\dots	$\mathcal{A}_n \times \mathcal{B}_0$
--------------------------------------	--	--	---------	--------------------------------------

Application les arbres binaires

Spécification récursive :

$$\mathcal{BinTree} = \text{Leaf} \sqcup \text{Node}(\mathcal{BinTree} \times \mathcal{BinTree})$$

Deux manières de compter les tailles :

1 Nombre de feuille :

$$\mathcal{BinTree} = \text{Leaf}_1 \sqcup \mathcal{BinTree} \times \mathcal{BinTree}$$

2 Nombre de Noeuds :

$$\mathcal{BinTree} = \text{Leaf}_0 \sqcup \text{Node}_1 \times \mathcal{BinTree} \times \mathcal{BinTree}$$

Application les arbres binaires

Spécification récursive :

$$\mathcal{BinTree} = \text{Leaf} \sqcup \text{Node}(\mathcal{BinTree} \times \mathcal{BinTree})$$

Deux manières de compter les tailles :

1 Nombre de feuille :

$$\mathcal{BinTree} = \text{Leaf}_1 \sqcup \mathcal{BinTree} \times \mathcal{BinTree}$$

2 Nombre de Noeuds :

$$\mathcal{BinTree} = \text{Leaf}_0 \sqcup \text{Node}_1 \times \mathcal{BinTree} \times \mathcal{BinTree}$$

Liste de tous les arbres à n Nœuds

Algorithme

- **Entrée** : un entier positif ou nul n
- **Sortie** : une liste d'arbres

```
if n == 0:  
    yield arbreVide()  
for i in range(n):  
    for g in BinTree(i):  
        for d in BinTree(n-1-i):  
            yield Noeud(g,d)
```

Nombre de Catalan

Proposition

Le nombre d'arbres binaires à n nœuds est appelé n -ième nombre de Catalan noté C_n . Les nombre de Catalan vérifient la récurrence :

$$C_0 = 1 \quad C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i} .$$

On peut trouver une formule close :

$$C_n = \frac{(2n)!}{n!(n+1)!} .$$

Voici les premières valeurs :

$$C_0 = 1, \quad C_1 = 1, \quad C_2 = 2, \quad C_3 = 5, \quad C_4 = 14, \quad C_5 = 42, \quad c_6 = 132 .$$

Specification d'une classe combinatoire

But : on veut décrire une **classe combinatoire** de manière à pouvoir appliquer automatiquement les algorithmes de comptage, itération, génération, . . .

Pour ceci, on va utiliser une **grammaire** qui code comment appliquer récursivement les constructions précédentes.

Specification d'une classe combinatoire

Retenir (Grammaire de description d'objets)

Constructeurs terminaux :

- $\mathcal{E}(o)$: la classe qui contient un seul objet o de taille 0
- $\mathcal{Z}(o)$: la classe qui contient un seul objet o de taille 1

Constructeurs binaires :

- $\mathcal{C} = \text{Union}(\mathcal{A}, \mathcal{B})$: union disjointe graduée :

$$|a|_{\mathcal{C}} = |a|_{\mathcal{A}} \text{ si } a \in \mathcal{A} \quad |a|_{\mathcal{C}} = |b|_{\mathcal{B}} \text{ si } b \in \mathcal{B}$$

- $\mathcal{C} = \text{Prod}(\mathcal{A}, \mathcal{B})$: produit cartésien graduée :

$$|(a, b)|_{\mathcal{C}} = |a|_{\mathcal{A}} + |b|_{\mathcal{B}} \text{ si } a \in \mathcal{A} \text{ et } b \in \mathcal{B}$$

Exemples :

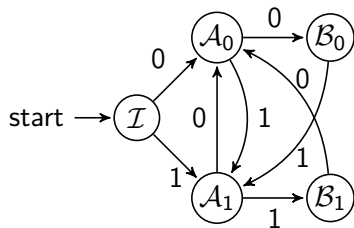
- Les arbres binaires, la taille est le nombres de **Noeuds**

$$\mathcal{BinTree} = \text{Union}(\mathcal{E}(\perp), \text{Prod}(\mathcal{Z}(\circ), \text{Prod}(\mathcal{BinTree}, \mathcal{BinTree})))$$

- Les arbres binaires, la taille est le nombres de **Feuilles**

$$\mathcal{BinTree} = \text{Union}(\mathcal{Z}(\perp), \text{Prod}(\mathcal{BinTree}, \mathcal{BinTree}))$$

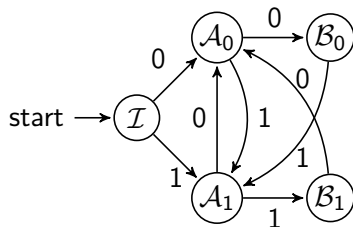
Exemples : Codage d'un automate fini



Les mots binaires qui n'ont pas plus de deux lettres identiques consécutives :

$$\begin{aligned} \mathcal{I} &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{A}_0) \sqcup (\mathcal{Z}("1") \times \mathcal{A}_1) \\ \mathcal{A}_0 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{B}_0) \sqcup (\mathcal{Z}("1") \times \mathcal{A}_1) \\ \mathcal{B}_0 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("1") \times \mathcal{A}_1) \\ \mathcal{A}_1 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{A}_0) \sqcup (\mathcal{Z}("1") \times \mathcal{B}_1) \\ \mathcal{B}_1 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{A}_0) \end{aligned}$$

Exemples : Codage d'un automate fini



Plus simple, on inclus la lettre dans l'état :

$$\mathcal{I} = \mathcal{E}(\varepsilon) \sqcup \mathcal{A}_0 \sqcup \mathcal{A}_1$$

$$\mathcal{A}_0 = \mathcal{Z}("0") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{B}_0 \sqcup \mathcal{A}_1)$$

$$\mathcal{B}_0 = \mathcal{Z}("0") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{A}_1)$$

$$\mathcal{A}_1 = \mathcal{Z}("1") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{B}_1 \sqcup \mathcal{A}_0)$$

$$\mathcal{B}_1 = \mathcal{Z}("1") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{A}_0)$$