Plus longue sous-séquence commune [dy02] Exercice

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - PLSSC (Solution)

Mots-Clés Programmation dynamique ■
Requis Axiomatique impérative, Récursivité des actions, Complexité des algorithmes ■
Difficulté ●●○ (3 h) ■

Objectif
Cet exercice calcule la longueur et trouve **une** plus longue sous-séquence commune à deux séquences A et B.

1 Plus longue sous-séquence commune / pgplssc

1.1 Reconnaissance d'une sous-séquence

Définition
Soit $A = (a_1, \ldots, a_n)$ une séquence de n éléments.
On appelle **sous-séquence** de A, toute séquence obtenue en retirant un nombre quelconque, éventuellement nul, d'éléments à A et en conservant l'ordre relatif des éléments restants.

Exemple
Si $A = (3, 1, 4, 1, 5, 9, 8)$ alors les séquences A, $(1, 1, 5)$, $C_1 = (1, 5, 9)$ et $()$ (la séquence vide) sont des sous-séquences de A, tandis que la séquence $(5, 1, 9)$ n'en est pas une. De même, si on considère les lettres minuscules, alors *bbcdd* est une sous-séquence de *aaabbbcccddd*.

Définissez le type `Sequence` comme étant un vecteur d'entiers caractérisant les séquences manipulées, ainsi que le type `ConstIterateur` de l'itérateur `const`ant sur une `Sequence`.

Outil C++
La classe générique `vector<T>` est définie dans la bibliothèque `<vector>`.

Validez vos définitions avec la solution.

Solution C++ @[pgplssc.cpp]

```
typedef vector<int> Sequence;
//typedef Sequence::iterator Iterateur;
typedef Sequence::const_iterator ConstIterateur;
```

Soient deux séquences $A = (a_1, \ldots, a_n)$ et $C = (c_1, \ldots, c_q)$.
Dites en français, comment reconnaître une sous-séquence C dans A.

Solution simple
On parcourt A et quand on trouve l'égalité avec l'élément courant de C, on le « retire » de C. A la fin, si C est vide, alors C est sous-séquence de A.

Écrivez une fonction `estSS1(A,C)` qui teste et renvoie `Vrai` si une `Sequence C` est une sous-séquence d'une `Sequence A`, `Faux` sinon.

Validez votre fonction avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
  Test de sous-sequence (version iterative)
  @param[in] a - une Sequence
  @param[in] c - une Sequence
  @return Vrai si C est sous-sequence de A
*/
bool estSS1(const Sequence& a, const Sequence& c)
{
  // Arret immediat si C est plus grand que A
  if (a.size() < c.size())
  {
    return false;
  }

  // Avance jusqu'a ce qu'une des sequences soit vide
  ConstIterateur it1 = a.begin();
  ConstIterateur it2 = c.begin();
  while (it1 != a.end() && it2 != c.end())
  {
    // Si egalite: avance dans C
    if (*it1 == *it2)
    {
      ++it2;
    }
    // Passe a l'element suivant dans A
    ++it1;
  }

  //ici si C est entierement traversee, alors OK
  return (it2 == c.end());
}
```

De même, écrivez une version récursive `estSS2(A,C)` du test de sous-séquence.

Aide méthodologique

Écrivez une fonction récursive `estSSRec(A,n,C,q)` qui teste et renvoie `Vrai| si une \lstinline`Sequence C@ (de longueur `q`) est une sous-séquence d'une `Sequence A` (de longueur `n`), `Faux` sinon.

Validez vos fonctions avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
  Test de sous-sequence (version recursive)
  @param[in] itA - iterateur de Sequence A
  @param[in] nA - taille de A
  @param[in] itC - iterateur de Sequence C
  @param[in] nC - taille de C
  @return Vrai si C est sous-sequence de A
*/
bool estSSRec(ConstIterateur itA, unsigned nA, ConstIterateur itC, unsigned nC)
```

```
{
  if (nC == 0)
  {
    return true;
  }
  else if (nA == 0)
  {
    return false;
  }
  else if (*itA == *itC)
  {
    return estSSRec(++itA, nA-1, ++itC, nC-1);
  }
  else
  {
    return estSSRec(++itA, nA-1, itC, nC);
  }
}
```

```
/**
  Test de sous-sequence (version recursive)
  @param[in] a - une Sequence
  @param[in] c - une Sequence
  @return Vrai si C est sous-sequence de A
*/
bool estSS2(const Sequence& a, const Sequence& c)
{
  if (a.size() < c.size())
  {
    return false;
  }
  else
  {
    return estSSRec(a.begin(),a.size(),c.begin(),c.size());
  }
}
```

Prouvez la version récursive de votre algorithme.

Solution simple

Lorsque C ou A est vide, la fonction renvoie immédiatement le résultat correct. Supposons C et A non vides. Si C est une sous-séquence de A, alors il existe une suite d'indices $1 \leq i_1 < i_2 < \ldots < i_q \leq n$ tels que $c_k = a_{i_k}$ pour tout k. Donc si $c_1 \neq a_1$, alors $i_1 \geq 2$ et C est une sous-séquence de $(a_2, \ldots, a_n)$. Lorsque $c_1 = a_1$, $(c_2, \ldots, c_q) = (a_{i_2}, \ldots, a_{i_q})$ est une sous-séquence de $(a_2, \ldots, a_n)$ car $i_2 \geq 2$.

Réciproquement, si $c_1 = a_1$ et si $(c_2, \ldots, c_q)$ est une sous-séquence de $(a_2, \ldots, a_n)$ alors il existe des indices $2 \leq i_2 < \ldots < i_q \leq n$ tels que $c_k = a_{i_k}$ pour $k \geq 2$, et cette relation a encore lieu pour $k = 1$ en posant $i_1 = 1$, donc C est une sous-séquence de A. Enfin, si $c_1 \neq a_1$ et C est une sous-séquence de $(a_2, \ldots, a_n)$ alors C est évidemment une sous-séquence de A.

Conclusion : Ceci prouve que si l'appel `estSSRec(A,n,C,q)` termine, alors il renvoie le bon résultat. Et il y a terminaison car n diminue d'une unité à chaque appel récursif.

En supposant que `C` est une sous-séquence de `A`,
Écrivez une fonction itérative `asupprimer1(A,C)` qui calcule et renvoie **une** liste `Sequence L` des indices (par rapport à 1) à supprimer d'une `Sequence A` pour obtenir une `Sequence C`. Lorsqu'il existe plusieurs listes `L` solution, une seule sera calculée. Par exemple, une solution pour `C1` est `L=(1,3,4,7)`.

Aide simple

Attention, La construction de la liste des indices n'est pas terminée lorsque `C` est vide car il faut constituer la liste des indices des éléments restant dans `A`.

Solution simple

On reprend l'algorithme précédent (version itérative) en accumulant les indices des éléments de `A` non appariés à ceux de `C`.

Validez votre fonction avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
  Liste des indices a supprimer (version iterative)
  @param[in] a - une Sequence
  @param[in] c - une Sequence
  @return liste des indices a supprimer de A pour obtenir C
*/
list<unsigned> asupprimer1(const Sequence& a, const Sequence& c)
{
  list<unsigned> result;

  // Avance tantque possible dans les sequences A et C
  unsigned ix = 1;
  ConstIterateur it1 = a.begin();
  ConstIterateur it2 = c.begin();
  while (it1 != a.end() and it2 != c.end())
  {
    if (*it1 == *it2)
    {
      ++it2;
    }
    else
    {
      result.push_back(ix);
    }

    ++it1;
    ++ix;
  }

  // si A n'est pas totalement traverse, ajoute les indices restants
  while (it1 != a.end())
  {
    result.push_back(ix);
```

```
    ++it1;
    ++ix;
  }

  return result;
}
```

De même, écrivez une version récursive `asupprimer2(A,C)` des indices à supprimer.

Aide méthodologique

Ici aussi écrivez une procédure récursive `asupprimerRec(A,n,C,q,L,m)` qui calcule **une** liste `Sequence L` des indices à supprimer d'une `Sequence A` (de longueur `n`) pour obtenir une `Sequence C` (de longueur `q`). L'entier `m` restituera la longueur de `L`.

Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsVector.cpp]

Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

```
#include "UtilsVector.cpp"
using namespace UtilsVector;
```

Soit la procédure générique `afficherVector(v)` qui affiche un `vector<T> v`.

C++ @[afficherVector] (dans UtilsVector.cpp)

Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsList.cpp]

Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

```
#include "UtilsList.cpp"
using namespace UtilsList;
```

Soit la procédure générique `afficherList(ls)` qui affiche une `list<T> ls`.

C++ @[afficherList] (dans UtilsList.cpp)

Écrivez une procédure `test_estSS` qui teste vos fonctions.

Testez. Exemple d'exécution :

```
A = [3 1 4 1 5 9 8 ]
C = [1 5 9 ]
estSS(A,C): iter= 1; rec= 1
Indices a supprimer: (1 3 4 7 )
A = [5 1 8 4 3 9 1 ]
C = [9 1 5 ]
estSS(A,C): iter= 0; rec= 0
A = [3 9 8 5 1 4 1 ]
C = [9 1 5 ]
estSS(A,C): iter= 0; rec= 0
```

Validez votre procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
void test_estSS()
{
  int valsA[] = {3,1,4,1,5,9,8};
  const unsigned nA = sizeof(valsA) / sizeof(int);
  int valsC[] = {1,5,9};
  const unsigned nC = sizeof(valsC) / sizeof(int);

  for (unsigned ix = 1; ix <= 3; ++ix)
  {
    Sequence A(&valsA[0], &valsA[nA]);
    Sequence C(&valsC[0], &valsC[nC]);
    bool b1 = estSS1(A, C);
    bool b2 = estSS2(A, C);
    cout<<"A = ";
    afficherVector(A);
    cout<<"C = ";
    afficherVector(C);
    cout<<"estSS(A,C): iter= "<<b1<<"; rec= "<<b2<<endl;
    if (b1 != b2)
    {
      cerr<<"OUPS... erreur dans estSS()"<<endl;
    }

    list<unsigned> ls;
    if (b1)
    {
      ls = asupprimer1(A, C);
      cout<<"Indices a supprimer: ";
      afficherList(ls);
    }

    random_shuffle(&valsA[0], &valsA[nA]);
    random_shuffle(&valsC[0], &valsC[nC]);
  }
}
```

1.2 Calcul de LC(A,B)

Définition
Soient $A = (a_1, \ldots, a_n)$ et $B = (b_1, \ldots, b_p)$ deux séquences.
On appelle **sous-séquence commune à A et B**, toute séquence $C = (c_1, \ldots, c_q)$ qui est à la fois une sous-séquence de A et de B.

Définition
Soient $A = (a_1, \ldots, a_n)$ et $B = (b_1, \ldots, b_p)$ deux séquences.
Une **plus longue sous-séquence commune à A et B** (en abrégé $PLSSC(A, B)$) est une sous-séquence commune de longueur maximale. La **longueur** d'une $PLSSC(A, B)$ est notée $LC(A, B)$.

Remarque
A et B peuvent avoir plusieurs PLSSC, mais elles ont toutes **même longueur**.

Exemple
Les PLSSC de $A = (5, 2, 4, 3, 1, 7)$ et $B = (5, 3, 2, 1, 6)$ sont $(5, 2, 1)$ et $(5, 3, 1)$.

Propriété
On note :

- $A_i = (a_1, \ldots, a_i)$ (avec $i \in [0..n]$).
- Et $B_j = (b_1, \ldots, b_j)$ (avec $j \in [0..p]$).

Soit $\ell_{i,j} = LC(A_i, B_j)$ avec la convention que $A_0 = B_0 = ()$.
Par définition :

$$\ell_{i,j} = \begin{cases} 1 + \ell_{i-1,j-1} & \text{si } a_i = b_j \\ \max(\ell_{i-1,j}, \ell_{i,j-1}) & \text{sinon} \end{cases} \qquad \text{(R)}$$

Prouvez l'équation récurrente (R).

Solution simple
Soit $C_k = (c_1, \ldots, c_k)$ une PLSSC de A_i et B_j.

- Si $a_i = b_j$ alors $c_k = a_i = b_j$ et C_{k-1} est une PLSSC de A_{i-1} et B_{j-1}
- Si $a_i \neq b_j$ et $c_k \neq a_i$ alors C_k est une PLSSC de A_{i-1} et B_j
- Si $a_i \neq b_j$ et $c_k \neq b_j$ alors C_k est une PLSSC de A_i et B_{j-1}

Soit le modèle de classes `Matrice<T>` des matrices d'éléments de type `T`.
Le modèle dispose de :

- Un constructeur qui prend deux entiers `n` (par défaut 1) et `p` (par défaut 1) et une valeur `T initval` (par défaut `T()`), crée une matrice de `n` lignes et `p` colonnes et initialise chaque élément à la valeur `initval`.

- Un accesseur `rows` du nombre de lignes.
- Un accesseur `cols` du nombre de colonnes.
- Un accesseur `get(ix,jx)` de la valeur de l'élément en ligne `ix`, colonne `jx`.
- Un modifieur `set(ix,jx,val)` qui fixe la valeur `val` de type `T` de l'élément en ligne `ix`, colonne `jx`.

C++ @[Matrice.hpp] @[Matrice.cpp]

Écrivez une fonction `LC(A,B)` qui, à partir des `Sequence A` (de longueur `n|) et \lstinline`Sequence B@ (de longueur `p`), calcule et renvoie la `Matrice<int>` des longueurs $L = (\ell_{ij})$ pour $i \in [0..n]$ et $j \in [0..p]$.

Aide méthodologique

Il faut générer toutes les valeurs de la $PLSSC(i,j)$ dans un ordre compatible avec l'équation (R). Pour calculer la $PLSSC(i,j)$ il faut disposer des trois éléments $PLSSC(i-1,j)$, $PLSSC(i,j-1)$ et $PLSSC(i-1,j-1)$. Il faut donc parcourir en lignes, en colonnes ou en anti-diagonales.

Prouvez la validité de votre algorithme.

Solution simple

A la suite de la création de L on a automatiquement $L_{0,j} = L_{i,0} = 0$ pour tous i et j. Ceci prouve que la propriété suivante est vérifiée lors de la première entrée dans l'itérative interne ($i = j = 1$) :

« tous les coefficients $L_{u,v}$ sont corrects si $u < i$ ou ($u = i$ et $v < j$) »

L'invariance de cette propriété résulte de ce que l'on applique les formules de la question qz1 et de ce que les valeurs $L_{i-1,j-1}$, $L_{i-1,j}$ et $L_{i,j-1}$ ont été correctement calculées auparavant.

Donnez la complexité asymptotique par rapport à n et p de l'opération `LC`.

Solution simple

Le temps de création de L est proportionnel à sa taille, soit $O(n\ p)$. Le temps de calcul d'un coefficient $L_{i,j}$ est borné (un accès à A, un accès à B, une comparaison, une addition ou un calcul de max et une affectation à un élément de tableau, plus quelques soustractions pour calculer $i-1$ et $j-1$) et chaque coefficient de L est calculé au plus une fois, donc le temps de remplissage est aussi $O(n\ p)$.

Calculez la matrice L sur l'exemple.

		5	3	2	1	6
	0	0	0	0	0	0
5	0	.	.	.	.	.
2	0	.	.	.	.	.
4	0	.	.	.	.	.
3	0	.	.	.	.	.
1	0	.	.	.	.	.
7	0	.	.	.	.	.

Solution simple

On obtient le tableau suivant :

		5	3	2	1	6
	0	0	0	0	0	0
5	0	1	1	1	1	1
2	0	1	1	2	2	2
4	0	1	1	2	2	2
3	0	1	2	2	2	2
1	0	1	2	2	3	3
7	0	1	2	2	3	3

Écrivez une procédure `PLSSC(A,B,L)` qui, pour des `Sequence A` et `Sequence B` et une `Matrice<int> L`, calcule et renvoie **une** `Sequence` $PLSSC(A,B)$.

Aide détaillée

Soit $q = L_{n,p}$ la longueur d'une PLSSC à A et B.

- Si $a_n = b_p$ alors d'après la question qz1, il existe une PLSSC à A et B obtenue en plaçant a_n au bout d'une PLSSC à A_{n-1} et B_{p-1}.
- Si $a_n \neq b_p$, alors on cherche une PLSSC à A_{n-1} et B_p ou a A_n et B_{p-1} selon que $L_{n-1,p} > L_{n,p-1}$ ou non.

Validez votre fonction et procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
  Matrice LC de deux sequences
  @param[in] a - une Sequence
  @param[in] b - une Sequence
  @return Matrice LC des sequences A et B
*/
Matrice<unsigned> LC(const Sequence& a, const Sequence& b)
{
  Matrice<unsigned> lc(a.size()+1, b.size()+1, 0);
```

```
  for (unsigned ii = 1; ii < lc.rows(); ++ii)
  {
    for (unsigned jj = 1; jj < lc.cols(); ++jj)
    {
      // ici tous les coefficients M[u,v] sont corrects...
      // ...si u < i ou si (u == i et v < j)
      int val = (a[ii-1] == b[jj-1])
              ? 1 + lc.get(ii-1, jj-1)
              : max(lc.get(ii-1, jj), lc.get(ii, jj-1));
      lc.set(ii, jj, val);
    }
  }
  return lc;
}
```

```
/**
  PLSC de deux sequences de matrice donnee
  @param[in] a - une Sequence
  @param[in] b - une Sequence
  @param[in] lc - Matrice des longueurs
  @return PLSC de A et B (de matrice LC)
*/
Sequence PLSC(const Sequence& a, const Sequence& b, const Matrice<unsigned>& lc)
{
  const unsigned n = a.size();
  const unsigned p = b.size();
  const unsigned q = lc.get(n, p);
  Sequence result(q);
  unsigned ii = n, jj = p, k = q;
  while (k != 0)
  {
    if (a[ii-1] == b[jj-1])
    {
      result[--k] = a[ii-1];
      --ii;
      --jj;
    }
    else if (lc.get(ii-1, jj) > lc.get(ii, jj-1))
    {
      --ii;
    }
    else
    {
      --jj;
    }
  }
  return result;
}
```

Téléchargez le fichier suivant et mettez-le dans votre dossier.

C++ @[UtilsMatrice.cpp]

Copiez/collez ensuite les lignes suivantes :

C++ **Au début** de votre programme :

```
#include "UtilsMatrice.cpp"
using namespace UtilsMatrice;
```

Soit le fichier suivant contenant la procédure générique `afficherMatrice(mt)` qui affiche une `Matrice<T> mt`.

C++ @[afficherMatrice] (dans UtilsMatrice.cpp)

Écrivez une procédure `test_calcul` qui teste sur la matrice exemple.

Testez. Résultat d'exécution :

```
A = [5 2 4 3 1 7 ]
B = [5 3 2 1 6 ]
cas vecteur:
0  0  0  0  0  0
0  1  1  1  1  1
0  1  1  2  2  2
0  1  1  2  2  2
0  1  2  2  2  2
0  1  2  2  3  3
0  1  2  2  3  3
une PLSC = [5 3 1 ]
```

Validez votre procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
void test_calcul()
{
  int valsA[] = {5,2,4,3,1,7};
  const unsigned nA = sizeof(valsA) / sizeof(int);
  int valsB[] = {5,3,2,1,6};
  const unsigned nB = sizeof(valsB) / sizeof(int);

  Sequence A(&valsA[0], &valsA[nA]);
  Sequence B(&valsB[0], &valsB[nB]);
  cout<<"A = ";
  afficherVector(A);
  cout<<"B = ";
  afficherVector(B);

  Matrice<unsigned> matV = LC(A, B);
  cout<<"cas vecteur:"<<endl;
  afficherMatrice(matV);
  Sequence s = PLSC(A, B, matV);
  cout<<"une PLSC = ";
  afficherVector(s);
}
```

1.3 Prétraitement de A et B, cardinal E petit

Pour accélérer le calcul d'une $PLSSC(A, B)$, on supprime de A tous les éléments ne figurant pas dans B et on supprime de B tous ceux ne figurant pas dans A. On note $\hat{A}$ et $\hat{B}$ les séquences obtenues.

Propriété

Toute $PLSSC(A, B)$ est une $PLSSC(\hat{A}, \hat{B})$ et réciproquement.

Exemple

$\hat{A}$ résulte de ce que ni 2 ni 3 ni 5 ne sont dans B, et $\hat{B}$ car 8 n'est pas dans A.

A	5	2	2	9	4	6	2	3	2	
B	8	9	6	4	9	6	4	4	8	8
$\hat{A}$	9	4	6							
$\hat{B}$	9	6	4	9	6	4	4			

Supposons d'abord que les éléments de A et B appartiennent à un ensemble E de cardinal suffisamment petit pour qu'il soit envisageable de créer en mémoire des listes ou des vecteurs de longueur $card(E)$. Ceci est le cas si A et B sont des séquences de caractères par exemple.

Écrivez une procédure efficace `pretraiter(A,B,Abis,Bbis,r)` qui, à partir des `Sequence A` et `Sequence B`, calcule les séquences $\hat{A}$ dans `Sequence Abis` et $\hat{B}$ dans `Sequence Bbis`. L'entier `r` désigne le cardinal de l'ensemble E.

Aide méthodologique

- Créez deux vecteurs booléens E_A et E_B indexés par les éléments de E (convertis en nombres entiers positifs au besoin) tels que : $E_A(x)$ est vrai ssi x apparaît dans A et $E_B(x)$ est vrai ssi x apparaît dans B.
- Ensuite parcourez A en extrayant les éléments x tels que $E_B(x)$ est vrai puis parcourez B en extrayant les éléments x tels que $E_A(x)$ est vrai.

Ceci donne les séquences $\hat{A}$ et $\hat{B}$.

Validez votre procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
 Pretraite les sequences A et B
 Calcul de Abis et Bbis
 @param[in] a - une Sequence
 @param[in] b - une Sequence
```

```
  @param[out] Abis - Sequence A-chapeau
  @param[out] Bbis - Sequence B-chapeau
  @param[in] r - cardinal de l'ensemble E
*/
void pretraiter(const Sequence& a, const Sequence& b,
                Sequence& Abis, Sequence& Bbis,
                unsigned r)
{
  // Cree les deux vecteurs d'apparition des elements
  // puis calcule les extraits
  Abis = extraitSeq(a, apparition(b, r));
  Bbis = extraitSeq(b, apparition(a, r));
}
```

Déterminez la complexité asymptotique du pré-traitement.

Solution simple

Soit r le cardinal de E. La création de E s'effectue en $O(r)$ opérations, le calcul de E en $O(n)$ opérations, et l'extraction de $\hat{A}$ à partir de A et E_B en $O(n)$ opérations. La complexité du pré-traitement est donc $O(n+p)$, c.-à-d. linéaire.

1.4 Prétraitement de A et B, cardinal E gros

Supposons maintenant que les éléments de A et B appartiennent à un ensemble E trop gros pour tenir en mémoire, mais muni d'une relation d'ordre total notée $\leq$. Ceci est le cas si A et B sont des séquences d'entiers par exemple.

Prétraitement de A et B

On suppose avoir trié A et B, c.-à-d. avoir calculé les séquences $A^t = (\sigma(1), \ldots, \sigma(n))$ et $B^t = (\tau(1), \ldots, \tau(p))$ où σ et τ sont des permutations de $[1..n]$ et $[1..p]$ respectivement telles que les séquences $(a_{\sigma(1)}, \ldots, a_{\sigma(n)})$ et $B = (b_{\tau(1)}, \ldots, b_{\tau(p)})$ sont croissantes.

Exemple

	1	2	3	4	5	6	7	8	9	10
A	5	2	2	9	4	6	2	3	2	
B	8	9	6	4	9	6	4	4	8	8
A^t	2	3	7	9	8	5	1	6	4	
B^t	4	7	8	3	6	1	9	10	2	5
dansA		T	T	T	T	T	T	T		
dansB				T	T	T				

Écrivez une procédure `pretraiter2(A,B,At,Bt,Abis,Bbis)` qui calcule les séquences $\hat{A}$ et $\hat{B}$ à partir des séquences A, B, A^t et B^t.

Aide détaillée

Passez en revue les séquences A^t et B^t comme pour une fusion, en notant les indices des éléments communs à A et B. Constituez alors les séquences $\hat{A}$ et $\hat{B}$ à partir de ces notes.

Validez votre procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
  Pretraite les sequences A et B (cas E gros)
  Calcul de Abis et Bbis
  @param[in] a - une Sequence
  @param[in] b - une Sequence
  @param[out] Abis - Sequence A-chapeau
  @param[out] Bbis - Sequence B-chapeau
  @param[in] r - cardinal de l'ensemble E
*/
void pretraiter2(const Sequence& a, const Sequence& b,
                 const vector<int>& at, const vector<int>& bt,
                 Sequence& Abis, Sequence& Bbis)
{
  const unsigned n = a.size();
  const unsigned p = b.size();
  vector<bool> dansB(n, false);
  vector<bool> dansA(p, false);

  // Calcule les vecteurs d'apparition
  unsigned ixA = 0;
  unsigned ixB = 0;
  while (ixA < n && ixB < p)
  {
    if (a[ at[ixA] ] == b[ bt[ixB] ])
    {
      // element commun: on prend sa valeur
      int x = a[ at[ixA] ];
      // Marque tous les elements de A egaux a cet element
      for (; ixA < n && a[ at[ixA] ] == x; ++ixA)
      {
        dansB[ at[ixA] ] = true;
      }
      // Marque tous les elements de B egaux a cet element
      for (; ixB < p && b[ bt[ixB] ] == x; ++ixB)
      {
        dansA[ bt[ixB] ] = true;
      }
    }
    else if (a[ at[ixA] ] < b[ bt[ixB] ])
    {
      ++ixA;
    }
    else
    {
      ++ixB;
    }
  }
```

Écrivez une procédure `test_pretraiter` qui teste sur les séquences exemple.

Testez. Résultat d'exécution :

```
A = [5 2 2 9 4 6 2 3 2 ]
B = [8 9 6 4 9 6 4 4 8 8 ]
extA_cas1 = [9 4 6 ]
extB_cas1 = [9 6 4 9 6 4 4 ]
extA_cas2 = [9 4 6 ]
extB_cas2 = [9 6 4 9 6 4 4 ]
```

Validez votre procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
void test_pretraiter()
{
  int valsA[] = {5,2,2,9,4,6,2,3,2};
  const unsigned nA = sizeof(valsA) / sizeof(int);
  int valsB[] = {8,9,6,4,9,6,4,4,8,8};
  const unsigned nB = sizeof(valsB) / sizeof(int);

  Sequence A(&valsA[0], &valsA[nA]);
  Sequence B(&valsB[0], &valsB[nB]);
  cout<<"A = ";
  afficherVector(A);
  cout<<"B = ";
  afficherVector(B);

  Sequence extA, extB;
  pretraiter(A, B, extA, extB, 256);
  cout<<"extA_cas1 = ";
  afficherVector(extA);
  cout<<"extB_cas1 = ";
  afficherVector(extB);

  int permA[] = {1,2,6,8,7,4,0,5,3}; //<- ATTENTION decalage de -1
  int permB[] = {3,6,7,2,5,0,8,9,1,4}; //resultant de l'offset 0
  Sequence at(&permA[0], &permA[nA]);
  Sequence bt(&permB[0], &permB[nB]);

  pretraiter2(A, B, at, bt, extA, extB);
  cout<<"extA_cas2 = ";
  afficherVector(extA);
  cout<<"extB_cas2 = ";
  afficherVector(extB);
}
```

Déterminez la complexité asymptotique du pré-traitement dans ce second cas. La complexité du tri devra être prise en compte : il n'est pas demandé d'écrire un algorithme de tri – vous pouvez citer un algorithme vu en cours.

Solution simple

Le calcul de A^t à partir de A se fait en $O(n \lg n)$ opérations si l'on emploie un algorithme de tri par fusion (le tri rapide QUICKSORT a aussi cette complexité, mais en moyenne seulement). Le calcul des vecteurs booléens E_A et E_B (notés dansA et dansB) a la complexité d'une fusion : $O(n+p)$ et l'extraction de $\hat{A}$ à partir de A et E_B se fait en $O(n)$ opérations. La complexité du pré-traitement est donc $O(n \cdot \lg n + p \cdot \lg p)$, c.-à-d. logarithmique.

Concluez : Est-il raisonnable d'effectuer un pré-traitement ? Justifiez la réponse.

Solution simple

En supposant $n = p$ et r petit devant n, on a une complexité de pré-traitement $O(n)$ ou $O(n \cdot \lg n)$, ce qui est négligeable devant la complexité de calcul de la PLSSC en $O(n^2)$. On peut donc toujours procéder au pré-traitement, ce n'est asymptotiquement pas pénalisant.

Cependant, l'utilité de ce pré-traitement est discutable. Dans le premier cas, si E est petit et les séquences A et B grandes, alors il est probable que presque tous les éléments de E figurent au moins une fois dans chacune des listes, et le pré-traitement ne diminue pas notablement les longueurs des séquences à traiter. Dans le second cas, si A et B sont aléatoires, indépendantes, et de longueur petite devant $card(E)$ alors elles sont probablement **presque disjointes** et le pré-traitement peut être efficace ; par contre, si A et B ne sont pas indépendantes (en particulier s'il s'agit de versions successives d'un fichier), il est probable que le pré-traitement ne permettra de retirer que quelques éléments seulement. Enfin, si n et p sont **petits**, le temps de pré-traitement ne peut plus être considéré comme négligeable : on déconseille le pré-traitement dans ce cas.

1.5 Extensions

On souhaite obtenir toutes les PLSSC des deux séquences. Pour résoudre ce problème, une idée est de calculer une matrice de traçage pour savoir quelle a été la relation utilisée (a, b ou c). Par exemple, on peut utiliser le fléchage suivant :

- $\nwarrow$ (a)
- $\uparrow$ (b)
- $\longleftarrow$ (c)
- # (quand $PLSSC(i-1,j) = PLSSC(i,j-1)$)

Pour avoir toutes les PLSSC, il faut alors suivre tous les chemins (ce qui explique la distinction entre $\nwarrow$ et # dans la fonction de calcul).

Calculez la matrice de traçage des séquences exemple.

		5	3	2	1	6
	0	0	0	0	0	0
5	0	.	.	.	.	.
2	0	.	.	.	.	.
4	0	.	.	.	.	.
3	0	.	.	.	.	.
1	0	.	.	.	.	.
7	0	.	.	.	.	.

Solution simple

On obtient le tableau suivant :

		5	3	2	1	6
	0	0	0	0	0	0
5	0	↖	←	←	←	←
2	0	↑	#	↖	←	←
4	0	↑	#	↑	#	#
3	0	↑	↖	#	#	#
1	0	↑	↑	#	↖	←
7	0	↑	↑	#	↑	#

Écrivez d'abord une fonction `LCdirections(A,B,lc)` qui calcule la matrice de traçage des `Sequence A` et `Sequence B` étant donnée également la `Matrice lc` des longueurs.

Déduisez une procédure récursive `afficherPLSCrec(A,B,lcdir)` qui affiche la PLSC des `Sequence A` et `Sequence B` étant donnée la `Matrice lcdir` des directions.

Aide simple

Partez du coin inférieur droit et suivez un chemin quelconque.

De même, écrivez une version itérative `afficherPLSC(A,B,lcdir)` de la PLSC.

Validez votre fonction et vos procédures avec la solution.

Solution C++ @[pgplssc.cpp]

```
/**
  Calcul de la matrice des directions
  Utilise la formule de recurrence
  @param[in] a - une Sequence
  @param[in] b - une Sequence
  @param[in] lc - Matrice des longueurs
  @return la matrice des directions
*/
enum{
  F_HAUT = char(94),    //direction flèche-haut
```

```
  F_GAUCHE = char(174), //flèche-a-gauche
  F_GH = '#',           //diagonale-egalite
  F_DIAG = char(92)     //flèche-diagonale
};
Matrice<char> LCdirections(const Sequence& a, const Sequence& b,
                           const Matrice<unsigned>& lc)
{
  Matrice<char> result(a.size()+1, b.size()+1, ' '); //'.'

  for (unsigned ii = 1; ii < lc.rows(); ++ii)
  {
    for (unsigned jj = 1; jj < lc.cols(); ++jj)
    {
      if (a[ii-1] == b[jj-1])
      {
        result.set(ii, jj, F_DIAG);
      }
      else
      {
        int diff = static_cast<int>(lc.get(ii-1, jj)) - lc.get(ii, jj-1);
        char val = (diff > 0 ? F_HAUT : diff < 0 ? F_GAUCHE : F_GH);
        result.set(ii, jj, val);
      }
    }
  }
  return result;
}
```

```
/**
  Affiche une PLSC issue des sequences A et B
  Version recursive
  @param[in] a - une Sequence
  @param[in] b - une Sequence
  @param[in] lcdir - Matrice des directions
  @param[in] ii - numero ligne
  @param[in] jj - numero colonne
*/
void afficherPLSCrec(const Sequence& a, const Sequence& b,
                     const Matrice<char>& lcdir,
                     unsigned ii, unsigned jj)
{
  // Arret en Haut a Gauche
  if (ii == 0 || jj == 0)
  {
    return;
  }
  // sinon cas generique
  switch (lcdir.get(ii, jj))
  {
    case F_GAUCHE:
      afficherPLSCrec(a, b, lcdir, ii, jj-1);
      break;
    case F_HAUT:
      afficherPLSCrec(a, b, lcdir, ii-1, jj);
      break;
    case F_GH:
      afficherPLSCrec(a, b, lcdir, ii-1, jj); //par exemple
      break;
```

```
    case F_DIAG:
      afficherPLSCrec(a, b, lcdir, ii-1, jj-1);
      cout<<"("<<a[ii-1] //affiche le couplet
          <<","<<b[jj-1] //...les deux elements devraient
          <<") ";          //...etre les memes (sinon BUG)
      break;
  }
}
```

```
/**
  Affiche une PLSC issue des sequences A et B
  Version Iterative
  @param[in] a - une Sequence
  @param[in] b - une Sequence
  @param[in] lcdir - Matrice des directions
  ATTENTION ici la PLSC est ecrite a l'envers!
*/
void afficherPLSC(const Sequence& a, const Sequence& b,
                  const Matrice<char>& lcdir)
{
  unsigned ii = lcdir.rows()-1;
  unsigned jj = lcdir.cols()-1;
  while (ii > 0 && jj > 0)
  {
    switch (lcdir.get(ii, jj))
    {
      case F_GAUCHE:
        --jj;
        break;
      case F_HAUT:
        --ii;
        break;
      case F_GH:
        --ii;
        break;
      case F_DIAG:
        cout<<"("<<a[ii-1]
            <<","<<b[jj-1]
            <<") ";
        --ii;
        --jj;
        break;
    }
  }
}
```

Écrivez une procédure `test_affich` qui teste sur les séquences exemple.

Testez. Résultat d'exécution :

```
A = [5 2 4 3 1 7 ]
B = [5 3 2 1 6 ]
cas vecteur:
0  0  0  0  0  0
0  1  1  1  1  1
```

```
0  1  1  2  2  2
0  1  1  2  2  2
0  1  2  2  2  2
0  1  2  2  3  3
0  1  2  2  3  3



   \  <  <  <  <
   ^  #  \  <  <
   ^  #  ^  #  #
   ^  \  #  #  #
   ^  ^  #  \  <
   ^  ^  #  ^  #

AfficherPLSCRec : (5,5) (2,2) (1,1)
AfficherPLSCIter: (1,1) (2,2) (5,5)
```

Validez votre procédure avec la solution.

Solution C++ @[pgplssc.cpp]

```
void test_affich()
{
  int valsA[] = {5,2,4,3,1,7};
  const unsigned nA = sizeof(valsA) / sizeof(int);
  int valsB[] = {5,3,2,1,6};
  const unsigned nB = sizeof(valsB) / sizeof(int);

  Sequence A(&valsA[0], &valsA[nA]);
  Sequence B(&valsB[0], &valsB[nB]);
  cout<<"A = ";
  afficherVector(A);
  cout<<"B = ";
  afficherVector(B);

  Matrice<unsigned> matV = LC(A, B);
  cout<<"cas vecteur:"<<endl;
  afficherMatrice(matV);
  cout<<endl;

  Matrice<char> matD = LCdirections(A, B, matV);
  afficherMatrice(matD);
  cout<<endl;

  cout<<"AfficherPLSCRec : ";
  afficherPLSCrec(A, B, matD, matD.rows()-1, matD.cols()-1);
  cout<<endl;
  cout<<"AfficherPLSCIter: ";
  afficherPLSC(A, B, matD);
  cout<<endl;
}
```

Calculez la complexité de la procédure récursive `afficherPLSCrec`.

Solution simple

La complexité est $O(n+p)$ puisque les indices i et j de a_i et b_j décroissent à chaque invocation.

Écrivez une fonction `nc(n,p)` qui calcule et renvoie le nombre maximum de $PLSSC(A,B)$. On pourra définir une version récursive.

Aide simple

Cette question revient à dénombrer le nombre de chemins reliant le point discret $(1,1)$ au point (n,p) en autorisant uniquement trois directions : $\longrightarrow$, $\nearrow$ et $\uparrow$.

1.6 Compléments

Ce problème a des applications pratiques dans les domaines (entre autres) :

Biologie moléculaire

Déterminer si deux chaînes d'ADN diffèrent « notablement », et sinon, identifier les différences.

Correction orthographique

Déceler les fautes d'orthographe dans un mot incorrect par comparaison avec les mots « voisins » figurant dans un dictionnaire.

Mise à jour de fichiers

A et B sont deux versions successives d'un fichier informatique ayant subi « peu » de modifications. Soit C une $PLSC(A,B)$. Pour diffuser B auprès des détenteurs de A, il suffit de transmettre les séquences des éléments de A et de B ne figurant pas dans C, ce qui est généralement plus économique que de transmettre B en totalité.

Commande diff

Le système UNIX possède une commande `diff` listant les différences entre deux fichiers texte suivant un algorithme inspiré de la recherche d'une PLSSC : La distance entre deux fichiers A et B considérés comme des listes de lignes est le nombre d'éléments de A et de B ne faisant pas partie d'une PLSSC à A et B. Elle est notée $d(A,B)$. On détermine cette distance en comparant A et B par *les deux bouts* : on progresse alternativement d'une unité de distance à partir des débuts de A et B, puis d'une unité à partir des fins de A et B, jusqu'à un point milieu, i.e. un couple (i,j) tel que $d(A_i,B_j) = d(A,B)/2$. Le nombre de couples (i,j) examinés au cours de cette recherche est majoré par $(n+p)\cdot d(A,B)$ et il n'est pas nécessaire de construire une matrice de taille $n \times p$ pour cela, il suffit de disposer de $O(d(A,B))$ positions mémoire. On peut alors obtenir récursivement une PLSSC à A

et B en concaténant une PLSSC à A_i et B_j avec une PLSC à $A \setminus A_i$ et $B \setminus B_j$. Le temps de calcul d'une PLSSC suivant cet algorithme (découvert indépendamment par Eugène W. Myers (1986) et E. Ukkonen (1985)) est en $O(n \cdot d(A, B))$ pour $n = p$. Pour en savoir plus, consulter l'aide EMACS sur la commande`diff` et le code source (en C) de GNU-diff.

2 Références générales

Comprend [Bajard-AL1 :c02 :ex06], [Quercia-AL1 :c12] ■