Série 11 :

Bases d'algorithmique

Buts

Cette série a pour but de vous faire approfondir la pratique de la programmation, sur le thème des bases d'algorithmique présentées en cours.

Préliminaires :

Avant de commencer les exercices décrits dans cette série, créez le répertoire ~/cpp/serie11 (i.e. créez le sous-répertoire serie11 dans le répertoire ~/cpp).
Travaillez dans ce repertoire.

Exercice 0 : reprise de l'exemple du cours (tableaux, niveau 0)

Cliquez ici si vous souhaitez faire cet exercice.

Exercice 1 : calcul de PGCD (algorithme d'Euclide, niveau 1)

Exercice n°38 (page 92 et 272) de l'ouvrage *C++ par la pratique*.
(PGCD = plus grand commun diviseur)

Buts

Écrivez le programme pgcd.cc qui :

1. demande à l'utilisateur d'entrer deux entiers strictement positifs a et b;
2. teste si a et b sont bien strictement positifs, et dans le cas contraire les redemande à l'utilisateur.
3. Trouve les entiers u, v et p satisfaisant l'identité de Bezout : u a + v b = p. Dans ce cas, on a p = pgcd(a,b)

Méthode

La méthode utilisée est **l'algorithme d'Euclide**.
On procédera par itération, comme suit (en notant x / y le quotient et x % y le reste de la division entière de x par y) :

0 : initialisation			$u_0 = 1$	$v_0 = 0$
	$x_1 = a$	$y_1 = b$	$u_1 = 0$	$v_1 = 1$
	...	...	...	...
i+1 : itération	$x_{i+1} = y_i$	$y_{i+1} = x_i \% y_i$	$u_{i+1} = u_{i-1} - u_i(x_i / y_i)$	$v_{i+1} = v_{i-1} - v_i(x_i / y_i)$
	...	...	...	...
Valeurs finales	x_{k-1}	y_{k-1} **!= 0**	u_{k-1}	v_{k-1}
k : condition d'arrêt quand $y_k = 0$	**$p = x_k$**	$y_k = 0$	inutile	inutile

C'est-à-dire que l'on va calculer de proche en proche les valeurs de x, y, u et v. En calculant à chaque fois les nouvelles valeur en fonction des anciennes (et en faisant bien attention de mémoriser ce qui est nécessaire à un calcul correct, voir les indications ci-dessous).
Par exemple, $y_{i+1} = x_i \% y_i$ veut dire : "la nouvelle valeur de y vaut l'ancienne valeur de x modulo l'ancienne valeur de y".
Programmez ces calculs dans une boucle, qui s'execute tant que la condition d'arrêt n'est pas vérifiée.
Pensez à initialiser correctement vos variables avant d'entrer dans la boucle.

Indications

Vu les dépendances entre les calculs, vous aurez besoin de définir (par exemple) les variables : x, y, u, v **et** q=x/y, r=x%y, prev_u, prev_v, new_u et new_v.
Vous mettrez ces variables à jour à chaque itération, à l'aide des formules de la ligne *i+1* et des relations temporelles évidentes entre elle (par exemple prev_u = u).
Testez que y est non nul avant d'effectuer les divisions !

Exemple d'execution

```
Entrez un nombre entier supérieur ou égal à 1 : 654321
Entrez un nombre entier supérieur ou égal à 1 : 210
Calcul du PGCD de 654321 et 210

  x         y         u         v
210       171         1     -3115
171        39        -1      3116
 39        15         5    -15579
```

```
 15          9        -11      34274
  9          6         16     -49853
  6          3        -27      84127
  3          0         70    -218107
PGCD(654321,210)=3
```

Notes

1. Ceux qui se sentent assez à l'aide peuvent le faire en utilisant la compilation séparée et réutiliser leur "librairie" demander_nombre.o. (voir par exemple l'exercice 5 de la série 10.
2. Remarquez que pour le seul calcul du pgcd, le calcul de x et y par l'algorithme ci-dessus suffit. Pas besoin de u et v. Ils osnt ici pour trouver l'équation de Bezout (et vous faire programmer des suites imbriquées)

Exercice 2 : Tranche maximale (structures, tableaux, algorithmique, niveau 1)

Positionnement du problème

On s'intéresse ici au problème suivant :
étant donnée une séquence (finie) de nombres entiers, positifs et négatifs, trouver la sous-séquence (contiguë) de somme maximale.
Par exemple si la séquence est :

```
11, 13, -4, 3, -26, 7, -13, 25, -2, 17, 5, -8, 1
```

la sous-séquence 3, -26, 7, -13, 25 a pour somme -4,
et la sous-séquence de somme maximale est

```
25, -2, 17, 5
```

(de somme 45).

Structures de données

Commençons par nous préoccuper des structures de données nécessaires pour résoudre informatiquement ce problème.
Il nous faut définir :

- comment représenter la séquence
  Un tableau (soit statique soit dynamique) semble bien approprié
- ce qu'est une sous-séquence
  De plus il faudra connaitre la somme correspondante pour décider si elle est maximale ou non.
  Pour cela je vous propose de coder une sous-séquence et sa somme dans une structure comprenant :
  - La position de début de la sous-séquence
  - La position de fin

- La somme correspondante

Cette première réflexion étant faite, on peut commencer.
Dans la version simple présentée ici, on définira dans un premier temps la séquence "en dur" dans le programme par un tableau de taille fixe.
Mais vous êtes libres de choisir une autre solution plus avancée si vous le souhaitez.

À faire :

Dans un fichier tranche.cc

1. Définissez le type Position comme un entier non-signé
2. (version statique) Déclarez une constante MAX de type Position valant, disons, 512.
3. Définissez le type Sequence comme un tableau statique d'entiers, de taille MAX
4. Définisez la structure SousSequence contenant 2 champs, debut et fin, de type Position, et un champs somme de type entier.
5. Dans le main(), déclarez la variable seq de type Sequence et initialisez la une valeur choisie, par exemple :
   `11, 13, -4, 3, -26, 7, -13, 25, -2, 17, 5, -8,  1`et déclarer la variable taille_seq de type position, initialisée à la taille réelle de la séquence seq (13 si vous avez pris l'exemple ci-dessus)

Algorithmes (et complexité)

Algorithme naïf

L'algorithme le plus simple qui vient à l'esprit est de chercher parmi toutes les sous-séquences, celle de somme maximale. "Chercher parmi toutes les sous-séquences" signifie, pour tous les débuts possibles, et pour toutes les fins possible pour un tel début.
D'où l'algorithme de base :

```
Entree : sequence de N nombres
Sortie : la sous-sequence (debut, fin, somme) de somme maximale

soussequence = (1, 1, sequence[1])
Pour tout debut de 1 à N
  Pour tout fin de debut à N
    somme = 0
    Pour tout position de debut à fin
      somme = somme + sequence[position]
    Si somme > soussequence.somme
      soussequence = (debut, fin, somme)
```

QUESTION : quelle est la complexité de cet algorithme ?
Implémentez cet algorithme (en C++) dans une fonction tranche_max_1 (de votre programme tranches.cc).
Complétez la fonction main() pour rechercher la sous-séquence (ou "tranche") de somme maximale dans la séquence précédemment considérée.

QUESTION : quelle est la réponse ?

Indications

1. Attention, dans les algorithmes les tableaux sont indexés de 1 à N, ce qui n'est pas le cas en C++...
2. Pensez à passer la taille de la sequence en argument de votre fonction.

Algorithme un peu moins naïf

La complexité de cet algorithme vient de ses boucles imbriquées.
Serait-il possible d'en enlever une ?
En regardant de plus prêt, on s'appercoit vite que l'on refait beaucoup de calculs plusieurs fois : le calcul de la somme d'une sous-séquence utilise souvent les calculs d'une sous-séquence précédente.
En d'autres termes, la boucle la plus intérieure est inutile car pour calculer la somme à "position+1", il suffit d'ajouter sequence[position+1] à l'ancienne somme.
Voici le nouvel algorithme :

```
Entree : sequence de N nombres
Sortie : la sous-sequence (debut, fin, somme) de somme maximale

soussequence = (1, 1, sequence[1])
Pour tout debut de 1 à N
  somme = 0
  Pour tout fin de debut à N
    somme = somme + sequence[fin]
    Si somme > soussequence.somme
      soussequence = (debut, fin, somme)
```

QUESTION : quelle est la complexité de cet algorithme ?
Implémentez cet algorithme dans une fonction tranche_max_2 et vérifiez qu'il fournit les bons résultats.

Algorithme linéaire

Peut-on aller plus loin ? Peut-on encore enlever une boucle ?
La réponse est **oui**, mais la solution est peut-être moins triviale.
L'idée reste cependant la même : supprimer des calculs inutiles. Mais elle utilise ici le fait que l'on cherche un maximum.
Si donc on trouve une sous-séquence initiale de somme inférieure (ou égale) à 0, on peut supprimer cette sous-séquence initiale car elle apporte moins à la somme totale que de commencer juste après elle.
Par exemple dans la séquence 4,-5,3,... il vaut mieux commencer au 3 (avec une somme initiale qui vaut 0) que commencer au 4 (qui nous donne une somme de -1 arrivé au 3).
D'où l'idée de l'algorithme suivant :

```
Entree : sequence de N nombres
Sortie : la sous-sequence (debut, fin, somme) de somme maximale
```

```
soussequence = (1, 1, sequence[1])
debut = 1
somme = 0
Pour tout fin de 1 à N
  somme = somme + sequence[fin]
  Si somme > soussequence.somme
    soussequence = (debut, fin, somme)
  Si somme <= 0
    debut = fin + 1
    somme = 0
```

QUESTION : quelle est la complexité de cet algorithme ?

Tests

Testez vos algorithmes avec les séquences suivantes :

```
-3, -4, -1, -2, -3
-1, -4, -3, -2, -3
 3, -1, -1, -1,  5
 3,  4,  1,  2, -3
 3,  4,  1,  2,  3
-5, -4,  1,  1,  1
```

Améliorations (FACULTATIF)

Vous pouvez aussi

1. vous aider d'une fonction affiche pour afficher les resultats
2. Faire saisir à l'utilisateur la séquence
3. Recommencer tant que l'utilisateur le souhaite
4. Lire des séquences dans un fichier

Exercice 3 : Profiling (algorithmique, niveau 2)

Cet exercice n'est pas difficile en soi. Il est niveau 2, uniquement parce qu'il s'intéresse à une notion "avancée" du cours : le profiling.
Reprendre l'exercice 2 de la série 10 sur les suites de Fibonacci (faites le si vous ne l'avez pas encore fait).
On veut ici regarder le temps mis par chaque fonction (profiling).
Pour cela compiler votre programme fibonacci.cc avec l'option -pg :

```
c++ -pg fibonacci.cc -o fibonacci
```

Ceux qui utilisent un Makefile, ajouter la ligne

`CPPFLAGS = -pg`
au début (cf cours sur la compilation séparée).
Lancez votre programme :
`fibonacci`
Faites le calculer une grosse valeur (par exemple 40).
Puis regardez les résultats :
`gprof fibonacci | more`
(Appuyez sur la barre espace pour avancer dans les pages)
ou si vous préférez lire un fichier :
`gprof fibonacci > fibonacci.prof`
puis ouvrez le fichier fibonacci.prof pour le lire.
Exemple de résultats : voir le cours.
Pour les détails sur le format des fichiers produits par gprof, lire la "manpage" de gprof :
`man gprof`

Exercice 4 : Machine de Turing (programmation, niveau 3)

Le but est d'écrire un programme turing.cc permettant de simuler une machine de Turing.
Si vous n'avez aucune idée sur la manière de procéder, vous pouvez utiliser la modélisation **proposée** ci-dessous, sinon allez y puis reprenez l'exercice ici.
Proposition de méthode :

1. Pour simuler une bande de longueur infinie, vous pouvez utiliser une string (définir par exemple un type Bande) :
   - lorsque l'on cherchera à déplacer la tête au-delà du dernier élément, il suffira de créer un élément supplémentaire, initialisé avec le symbole 'epsilon' (le caractère correspondant, à définir).
   - lorsque l'on cherchera à déplacer la tête avant le premier élément, il suffira d'insérer en début de chaîne le symbole 'epsilon'.
2. La tête de lecture sera représentée simplement par un entier, indiquant sa position courante. Je vous conseille aussi de définir un type ici, par exemple Tete.
3. Créez ensuite les fonctions implémentant les primitives de lecture, écriture et positionnement de la tête de lecture :
   char lecture(Bande& b, Tete& t); *lit le caractère 'courant'*
   void ecriture(char c, Bande& b, Tete& t); *remplace le caractère courant par la valeur transmise en argument*
   void avance(Tete& t); *déplace la tête de lecture vers la droite*
   void recule(Tete& t); *déplace la tête de lecture vers la gauche*
   Notez que, dans l'implémentation proposée, lecture et ecriture peuvent changer la valeur de tete : on simulera ainsi le fait que la bande est infinie, en insérant le bon nombre d'epsilon en début de bande lorsque la position de la tête est négative (naturellement,

lorsque les espilons auront étés insérés, on devra repositionner la tête à zéro). [on aurait également pu choisir d'effectuer ces opérations - simuler une bande infinie - lors des déplacements de la tête...]

4. Définissez une fonction initialise(Bande& b, string valeur) permettant d'initialiser la bande au moyen d'une chaîne de caractères
5. Pour représenter la table de transitions, utilisez un tableau statique d'au plus, disons, 512 lignes de transitions :
   - chaque 'case' définissant une transition est une structure de 3 champs ([prochain état, caractère, déplacement]);
   - chaque ligne de transition est un tableau de 3 'cases', une par valeur possible du caractère courant ([0, 1, epsilon]);

```
struct Transition
{
  Etat etat;
  char caractere;
  int deplacement;
}
typedef Transition Ligne[3]; // [ 0 , 1 , epsilon ]
```

   Avec une telle représentation, une table de transitions sera simplement un tableau de (au plus 512) Lignes.
6. Écrivez une structure **TMachine**, simulant une machine de Turing.
   Vous aurez besoin des champs suivants :
   - un entier (ou un type prédéfini par vous) modélisant l'**état actuel** de la machine;
   - une bande ;
   - une tête de lecture
   - une table de transitions
7. Définissez les fonctions suivantes :
   - une fonction permettant de créer une machine de Turing
     void cree(TMachine& m, const TableTransition& table);
   - une fonction permettant de ré-initialiser la machine (pour recommencer avec une nouvelle entrée)
     void reinitialise(TMachine& m, const string& entree);
   - une fonction permettant de démarrer la simulation de la machine.
     Choisissez une convention sur la valeur de l'état pour stopper la machine (par exemple, une valeur négative).

Application 1: Test de Parité

Testez votre machine avec l'exemple du cours testant la parité de l'entrée.
Avec la modélisation proposée, utilisez la syntaxe suivante pour définir une table de transition (dans la fonction cree :

```
  TableTransition table = {
    { {    1,     '0', +1}, {    1,     '1', +1}, {    2, EPSILON, -1}
```

```
},
    { {    3, EPSILON, -1}, {    4, EPSILON, -1}, {UNDEF, EPSILON, -1}
},
    { {    3, EPSILON, -1}, {    3, EPSILON, -1}, {    5,     '1', +1}
},
    { {    4, EPSILON, -1}, {    4, EPSILON, -1}, {    5,     '0', +1}
},
    { {UNDEF, EPSILON, -1}, {UNDEF, EPSILON, -1}, {UNDEF, EPSILON, -1}
}
  };
```

La constante UNDEF représente un état non défini également utilisé pour "l'ordre de fin d'exécution".
Dans cet exemple, on suppose que le déplacement vers l'avant est représenté par la valeur +1, et le déplacement vers l'arrière par -1.
Exemple (détaillé) d'execution :

```
Lancement de la machine de Turing
    --------------------------------
    Etat actuel : 1
    Bande :
       10
       ^ (Tete : 0)
    --------------------------------
 lu : 1, nouvel état : 1, écrit : 1, avance
    --------------------------------
    Etat actuel : 1
    Bande :
       10
        ^ (Tete : 1)
    --------------------------------
 lu : 0, nouvel état : 1, écrit : 0, avance
    --------------------------------
    Etat actuel : 1
    Bande :
       10
         ^ (Tete : 2)
    --------------------------------
 lu : e, nouvel état : 2, écrit : e, recule
    --------------------------------
    Etat actuel : 2
    Bande :
       10e
        ^ (Tete : 1)
    --------------------------------
 lu : 0, nouvel état : 3, écrit : e, recule
    --------------------------------
    Etat actuel : 3
    Bande :
```

```
      1ee
      ^ (Tete : 0)
   -------------------------------
 lu : 1, nouvel état : 3, écrit : e, recule
   -------------------------------
   Etat actuel : 3
   Bande :
      eee
     ^ (Tete : -1)
   -------------------------------
 lu : e, nouvel état : 5, écrit : 1, avance
   -------------------------------
   Etat actuel : 5
   Bande :
      1eee
       ^ (Tete : 1)
   -------------------------------
 lu : e, nouvel état : 0, écrit : e, recule
   -------------------------------
   Etat actuel : 0
   Bande :
      1eee
      ^ (Tete : 0)
   -------------------------------
Arrêt de la machine de Turing.
```

Application 2: Inversion de séquence sur une Machine de Turing

Écrivez (sur papier) la table de transition d'une machine de Turing réalisant l'inversion d'une séquence de bits.
Par exemple, si l'on a 1101 en entrée, on devra avoir 1011 en sortie.
On supposera que:

- la bande (de longueur infinie) ne contient rien d'autre que la séquence (i.e. à l'exception de la séquence, tous les caractères sont des 'epsilon' -> la séquence est encadrée [délimitée] par des 'epsilon');

- la tête de lecture/écriture est initialement positionnée sur le premier caractère de la séquence.

Remarquez bien qu'il n'est pas nécessaire que la séquence inversée occupe, sur la bande, la même position que la séquence initiale.

Testez votre machine sur un exemple simple (du moins dans un premier temps) !
Indications: si vous êtes en panne d'inspiration, vous pouvez essayer l'algorithme **proposé** ci-après.

1. Si la séquence d'entrée est vide, terminer. Sinon, aller jusqu'à la cellule immédiatement à droite de la frontière droite de la séquence, en mémorisant (dans la valeur des états : voir

l'algorithme de parité ci-dessus) la valeur du dernier élément de la séquence.

2. Si la (nouvelle) séquence n'est pas vide, aller tout d'abord à sa frontière droite (toujours en mémorisant le dernier élément), puis écrire l'élément mémorisé (qui est alors le dernier élément de la nouvelle séquence), se déplacer à la fin de la séquence d'entrée courante, effacer son dernier élément et se déplacer à gauche. (Si la séquence n'est pas vide, on est alors de nouveau sur le dernier élément de la séquence d'entrée 'courante')

3. Si la séquence d'entrée courante est vide, aller à la frontière gauche de la nouvelle séquence et terminer. Sinon, aller au début de la nouvelle séquence en mémorisant le dernier caractère de la séquence d'entrée courante, puis recommencer en (2)

Dernière mise à jour : 2005/12/07 14:18:20 (1.16)