Algo (CS101)

Exercise book

Table des matières

TD 1

Algorithmique autour des sommes

Objectifs

Conception de fonctions simples (itératives), en utilisant le langage support C. L'objectif est une pratique algorithmique plus qu'une pratique de la syntaxe C.

1.1 Fonctions itératives, boucles for

EXERCICE #1 ▸ **Puissance**
Écrire une fonction qui calcule x^n, x et n étant passés en paramètre.

EXERCICE #2 ▸ **Suite des suites**
Écrire une fonction permettant de calculer le n-ième terme de la suite définie par $u_0 = A$, $u_{n+1} = B * u_n + 1515$, A, B étant passés en paramètre de la fonction.

EXERCICE #3 ▸ **Sommes**
Soit N donné. Écrire des fonctions (avec N passé en paramètre), qui calculent et renvoient (on considère que la fonction puissance existe) :

1. $\sum_{i=1}^{N} i^3$.
2. $\sum_{i=1}^{N} \sum_{j=1}^{N} (i+j)^3$.
3. $\sum_{i=1}^{N} \sum_{j=i}^{N} (i+j)^3$.

1.2 Boucles while

EXERCICE #4 ▸ **Le premier tel que**

Concevoir une fonction qui calcule et retourne le plus petit entier n tel que $\sum_{i=1}^{n} \left(\sum_{j=1}^{i} (i+j) \right) \geq 1000$.

EXERCICE #5 ▸ **Calcul de valeur approchée de Pi par somme de série**

"On sait que" $\frac{\pi}{4} = arctan(1) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2*n+1}$.

Soit ε un réél fixé petit (10^{-5} par exemple). Écrire une fonction qui calcule une valeur approchée de π à ε près. On calculera des termes successifs qui approchent π et on s'arrêtera lorsque deux termes successifs auront une différence inférieure à ε.

TD 2

Algorithmique II : boucles, complexité, invariants.

Objectifs Manipuler les boucles simples et imbriquées. Les utiliser pour parcourir des tableaux. Premiers calculs de complexité, premiers invariants.

2.1 Boucles imbriquées

EXERCICE #1 ▸ **Le premier tel que**

— Comment calculer : $\sum_{j=1}^{i}(i+j)$ lorsque i est fixé ? Quel est le nombre d'opérations "somme" effectué ?

— Comment calculer : $\sum_{i=1}^{n}\big(\sum_{j=1}^{i}(i+j)\big)$ lorsque n est fixé ? Combien de sommes sont-elles effectuées ?

— Concevoir une fonction qui calcule et retourne le plus petit entier n tel que $\sum_{i=1}^{n}\big(\sum_{j=1}^{i}(i+j)\big) \geq 1000$.

2.2 Parcours de tableaux et invariants

EXERCICE #2 ▸ **Preuve d'invariant par induction**
Dans le cours nous avons montré que pour prouver la correction de l'algorithme "simple" de calcul de x^n (cf slides du cours), il fallait montrer $P(i)$: "à la fin de la boucle i, la variable locale contient x^i". Terminer proprement cette preuve.

EXERCICE #3 ▸ **Maximum d'un tableau**
Rappeler rapidement l'algorithme de recherche d'un élément dans un tableau d'entiers (cf le cours).

— Trouver un invariant de la boucle for et prouver cet invariant.

— Terminer la preuve de correction.

EXERCICE #4 ▸ **Palindrôme**
Écrire une fonction :

```
bool is_pal(char ch[N], int len)
```

qui retourne `true` si la chaîne ch (supposée de longueur len), est un palindrôme, `false` sinon. À l'aide d'un invariant que vous trouverez vous-même, prouvez la correction de cet algorithme.

EXERCICE #5 ▸ **Recherche d'un élément dans un tableau**
Prouver la correction de la recherche d'un élément dans un tableau avec rupture prématurée du flot.

EXERCICE #6 ▸ **Recherche dans un tableau trié**
Écrire un algorithmique (itératif) qui recherche dans un tableau supposé trié, par dichotomie :

```
bool recherche_dicho2(int t[N], int e):
// Précondition N>1, et pour tout i verifiant 0 <=i < N-2, on a t[i]<=t[i+1]
```

Ensuite, trouver un invariant de la boucle while (ie suffisant pour prouver la fonction), et le prouver.

TD 3

Algorithmique III : algorithmes de tableaux

Objectifs Décrire et concevoir des algorithmes de tableaux. Pratiquer les preuves de correction.

3.1 Recherche dans un tableau

EXERCICE #1 ▸ **Recherche d'un élément dans un tableau**
Prouver la correction de la recherche d'un élément dans un tableau avec rupture prématurée du flot, que l'on a réalisée en cours :

```
bool recherche1(int t[N], int el){
  for (int i=0; i<N; i++){
    if (t[i]==el) return true;
  }
  return false;
}
```

EXERCICE #2 ▸ **Recherche dans un tableau trié**
Écrire un algorithmique (itératif) qui recherche dans un tableau supposé trié, par dichotomie :

```
bool recherche_dicho2(int t[N], int e){
// Précondition N>1, et pour tout i vérifiant 0 <= i < N-2, on a t[i]<=t[i+1]
```

Ensuite, trouver un invariant de la boucle while (ie suffisant pour prouver la fonction), et le prouver.

EXERCICE #3 ▸ **À rédiger et à rendre sur Chamilo**
Rédiger dans un fichier texte ou sur papier et déposer sur Chamilo (utiliser un logiciel qui génère un pdf à partir d'une photo, si vous n'avez pas de scanner) avant jeudi 10/11 18H :

1. Modifier l'algorithme de recherche simple pour qu'il renvoie l'indice de la case où l'élément a été trouvé, -1 si l'élément n'a pas été trouvé. Rédiger la preuve de correction complètement.
2. Modifier l'algorithme de recherche dichotomique de la même façon. Qu'est-ce qui garantit que l'indice trouvé est le plus petit possible ?
3. (Bonus) Fournir un algorithme récursif pour la recherche dichotomique. Expliquer.

Travail personnel uniquement.

3.2 Un algorithme classique

On rappelle qu'un nombre premier n'est divisible uniquement par deux entiers distincts, 1 et lui-même. Ni 0 ni 1 ne sont premiers.

EXERCICE #4 ▸ **Crible**
Déterminez tous les nombres premiers inférieurs à un entier n donné par la méthode du crible d'Eratosthene : Cette méthode consiste à :

— Construire un tableau contenant tous les entiers de 0 à n.
— Barrer les éléments de ce tableau qui sont multiples de 2.
— Barrer les multiples de 3
— etc (la condition d'arrêt est à déterminer)

Allez voir la jolie animation sur la page Wikipédia!

TD 4

Algorithmique IV : algorithmes de chaînes

Objectifs Décrire et concevoir des algorithmes de tableaux. On commencera par l'exercice du crible d'Ératosthène du TD précédent.

4.1 Algorithmes élémentaires de chaînes

EXERCICE #1 ▸ **Miroir**
Soit M un mot de taille inconnue. Écrire un algorithme inversant le mot. On suppose l'existence d'une fonction `longueur` qui renvoie la longueur d'une chaîne de caractères passée en paramètre.

EXERCICE #2 ▸ **Nombre d'occurrences des lettres latines dans un texte**

Pour réaliser des statistiques sur l'usage des caractères dans des textes, on demande de réaliser une fonction qui calcule le nombre d'occurrences des lettres de l'alphabet latin (de 'a' à 'z', majuscules, minuscules), dans un texte. Le texte (qui ne contient que des lettres latines, et des espaces, aucun autre caractère) est considéré dans un tableau de taille statique.

On pourra utiliser une fonction `char to_lower(char t)` telle que :

```
to_lower('A') == to_lower('a') == 'a'
```

EXERCICE #3 ▸ **Calcul de la distance de Hamming**
Écrire une fonction qui calcule la distance de Hamming entre deux mots de même taille. La distance de Hamming entre "ramer" et "cases" est 3, car 3 caractères de même indice sont distincts.

EXERCICE #4 ▸ **Chiffrement césar**
Le chiffrement César d'un texte avec une clé k consiste à décaler chacun des caractères latins du texte de k. Concevoir une fonction qui permet de réaliser ce codage césar d'un texte. Explicitez vos hypothèses.

EXERCICE #5 ▸ **Recherche naïve de sous-chaîne dans un texte**
Concevoir une fonction qui recherche toutes les occurrences d'un mot (supposé de taille <N) dans un texte (supposé de taille <M, mais largement plus grand que N). Combien de comparaisons effectue votre fonction au pire ? Au mieux ?

Expliquer informellement avec un exemple comment on pourrait gagner un peu de comparaisons.

TD 5
Algorithmes de chaînes

About Ce problème est une annale d'examen (posé à Lille en 2011). Les exercices demandant du pseudo code seront effectués en pseudo code ou en C au choix.

Dans ce problème, nous allons coder une partie de l'algorithme de compression `bzip`, en fait la partie préparatoire du texte, qui consiste à faire apparaître des caractères identiques dans le texte à compresser. La partie compression des données n'est pas abordée.

1 - Transformation de Burrows-Wheeler

Cette transformation réalise une permutation des lettres du mot d'entrée (c'est-à-dire que toutes les lettres du mot d'entrée s'y retrouveront). Cette transformation a deux caractéristiques :

— dans le mot de sortie les répétitions de lettres identiques sont plus fréquentes.

— elle est bijective, c'est-à-dire qu'il existe une transformation inverse qui calcule à partir du mot transformé, le mot initial.

L'objet de cette partie est de calculer le codage d'un mot quelconque par la transformée.

Soit w un mot quelconque de taille ℓ, par exemple `banane`, qui est de taille 6. Dans un premier temps, on ajoute un marqueur de fin de mot, ♯ (on considère que ♯ n'est pas un caractère du mot). Ensuite, on construit une matrice qui contient toutes les permutations **circulaires** de ce texte, en décalant successivement le mot d'entrée vers la droite (matrice de gauche du dessin5.1).

$$\begin{pmatrix} b & a & n & a & n & e & \sharp \\ \sharp & b & a & n & a & n & e \\ e & \sharp & b & a & n & a & n \\ n & e & \sharp & b & a & n & a \\ a & n & e & \sharp & b & a & n \\ n & a & n & e & \sharp & b & a \\ a & n & a & n & e & \sharp & b \end{pmatrix} \qquad \begin{pmatrix} \sharp & b & a & n & a & n & \mathbf{e} \\ a & n & a & n & e & \sharp & \mathbf{b} \\ a & n & e & \sharp & b & a & \mathbf{n} \\ b & a & n & a & n & e & \mathbf{\sharp} \\ e & \sharp & b & a & n & a & \mathbf{n} \\ n & a & n & e & \sharp & b & \mathbf{a} \\ n & e & \sharp & b & a & n & \mathbf{a} \end{pmatrix}$$

FIGURE 5.1 – Matrice des permutations circulaires, puis la même, triée, et en gras le mot résultat.

Ensuite, les **lignes** de la matrice ainsi obtenue sont triées par ordre alphabétique (ordre du dictionnaire), en considérant les lignes comme des mots, ♯ étant inférieur à tous les autres caractères. Notre matrice devient alors la matrice de droite de la figure 5.1. (En effet, la ligne 1 : `anane#b` est inférieure à la ligne 2 : `ane#ban` car les deux premiers caractères sont identiques et le caractère 'a' est inférieur au caractère 'e'.)
Enfin, le mot résultat de la transformation est la dernière colonne de la matrice, ici en **gras**, c'est-à-dire `ebn#naa`.

Tous les tableaux de caractère (chaînes) auront une taille fixée à N (constante), N considéré toujours plus grand que la taille du mot à coder. Les tableaux de chaînes de caractères seront des matrices de caractères de taille $N \times N$. On rappelle qu'une chaîne de caractère possède un marqueur de fin, le caractère \0.

EXERCICE #1 ▸ **Questions simples, à préparer en avance**

1. Soit w le mot suivant (pour cette question on suppose N=9) :

h	e	l	l	o	\ 0			

Quelle est la taille du mot w ?

2. Calculer la transformée du mot `ima`.*On demande les calculs et le résultat, pas un algorithme*
3. Écrire en C une fonction `taille` qui prend en entrée une chaîne de caractères et qui renvoie sa taille.
4. Écrire en C une action `copie_tab` qui prend en paramètre deux chaînes de caractères, un entier ℓ, et qui copie la première chaîne dans la seconde sachant que la taille du mot est ℓ. *On n'oubliera pas de mettre le caractère de fin de chaîne à la bonne place.*
5. Écrire en C une action `decalage_droite` qui prend en entrée deux chaînes de caractères, un entier ℓ (la taille de la première chaîne), et qui modifie la deuxième chaîne de façon à ce qu'elle contienne la première chaîne décalée vers la droite. *Sur notre exemple, le décalé de* `banane#` *est* `#banane`, *le décalé de* `#banane` *est* `e#banan`.
6. Écrire en C une action `imprime_mat_carree` qui étant donnés une matrice mat de caractères de taille $N \times N$, et un entier ℓ imprime les caractères $mat[i][j]$ avec $0 \leq i \leq \ell - 1$ et $0 \leq j \leq \ell - 1$. *Cette fonction nous sera bien utile, car on va encoder des mots de taille quelconque dans des lignes de taille N, et on ne veut pas imprimer des caractères en trop.*
7. Écrire en C une action `mat_permut` qui étant donnés un mot d'entrée t, une matrice mat de caractères de taille $N \times N$, et un entier ℓ (taille du mot d'entrée), remplit la matrice mat avec les permutations du mot t. *On pourra s'apercevoir que $mat[i]$ est un tableau de taille N, et il est fortement recommandé d'utiliser les actions/fonctions précédentes.*
8. Écrire un programme **C complet** : main, fonctions (déclarations, et code avec des pointillés) , etc :
   — Inclusion des librairies `stdio.h` et `stdbool.h`.
   — Déclaration de la constante N valant 60
   — Déclaration et initialisation d'un tableau t contenant le mot `banane#`
   — Déclaration, construction et impression de la matrice de permutation de t.

EXERCICE #2 ▶ **Complexité du codage**

Quelle est la complexité en nombre d'affectations de votre algorithme de construction de la matrice de permutations, en fonction de ℓ taille du mot initial ?

Maintenant, nous allons trier la matrice de permutation (en place, sans utilisation d'une matrice auxiliaire), par lignes, selon l'ordre alphabétique, toujours en considérant que *les lignes sont des mots.*

EXERCICE #3 ▶ **Tri de la matrice de permutation**

1. **En pseudo-code**, écrire une fonction `comp_tab_alpha` qui prend en argument deux chaînes de caractères de taille ℓ, ainsi que l'entier ℓ et qui retourne :
   — $r = 0$ si les deux chaînes sont identiques
   — $r = -1$ si la première chaîne est strictement inférieure (au sens de l'ordre alphabétique) à la deuxième chaîne.
   — $r = 1$ sinon.
2. **En pseudo-code**, écrire une action `echange_lignes` qui prend en paramètres une matrice mat de caractères de taille $N \times N$, deux entiers $i1$ et $i2$, un entier ℓ, et qui échange le contenu des lignes $i1$ et $i2$ de la matrice mat (en considérant que seules les cases d'indice 1 à $\ell - 1$ doivent être copiées dans une ligne). *On utilisera la fonction* `copie_tab` *de la question 4.*
3. En utilisant les deux fonctions précédentes, écrire un algorithme **en pseudo-code** qui étant donné une matrice mat de caractères de taille $N \times N$, ainsi que l'entier ℓ donnant la taille réelle de la sous-matrice à trier, trie la matrice en lignes par ordre alphabétique. *On modifiera un tri connu, par exemple le tri sélection, pour l'adapter au cas matriciel.*
4. Quelle est la complexité de votre algorithme précédent en nombre d'affectations ?
5. À l'aide de toutes les fonctions/actions écrites précédemment, écrire un algorithme **en pseudo-code** qui réalise le codage d'un mot initial quelconque (initialement sans marqueur de fin ♯). Quelle est la complexité de cet algorithme en nombre d'affectations ?

2 - Transformation de Burrows-Wheeler inverse

L'objet de cette partie est de calculer la transformée inverse. L'algorithme consiste à construire une matrice à l'aide du mot codé (voir la figure 5.2 pour les étapes) :

— On trie les lettres du mot codé (ici donc, `ebn\#naa`) par ordre alphabétique et on le stocke dans la colonne 0 de la matrice à construire.

— On ajoute à gauche de la matrice (en décalant la colonne déjà construite) le mot codé (étape (a) sur la figure). On trie par ordre alphabétique les *lignes de* la matrice obtenue. (étape (t))

— On recommence jusqu'à remplissage complet de la matrice. Le résultat (le mot décodé) est lisible sur la première ligne de la matrice.

$$\begin{pmatrix} \sharp & \cdot & \dots \\ a & \cdot & \dots \\ a & \cdot & \dots \\ b & \cdot & \dots \\ e & \cdot & \dots \\ n & \cdot & \dots \\ n & \cdot & \dots \end{pmatrix} \xrightarrow[(a)]{} \begin{pmatrix} e & \sharp & \dots \\ b & a & \dots \\ n & a & \dots \\ \sharp & b & \dots \\ n & e & \dots \\ a & n & \dots \\ a & n & \dots \end{pmatrix} \xrightarrow[(t)]{} \begin{pmatrix} \sharp & b & \dots \\ a & n & \dots \\ a & n & \dots \\ b & a & \dots \\ e & \sharp & \dots \\ n & a & \dots \\ n & e & \dots \end{pmatrix} \xrightarrow[(a)]{} \begin{pmatrix} e & \sharp & b & \dots \\ b & a & n & \dots \\ n & a & n & \dots \\ \sharp & b & a & \dots \\ n & e & \sharp & \dots \\ a & n & a & \dots \\ a & n & e & \dots \end{pmatrix} \xrightarrow[\dots]{} \begin{pmatrix} \sharp & \boldsymbol{b} & \boldsymbol{a} & \boldsymbol{n} & \boldsymbol{a} & \boldsymbol{n} & \boldsymbol{e} \\ a & n & a & n & e & \sharp & b \\ a & n & e & \sharp & b & a & n \\ b & a & n & a & n & e & \sharp \\ e & \sharp & b & a & n & a & n \\ n & a & n & e & \sharp & b & a \\ n & e & \sharp & b & a & n & a \end{pmatrix}$$

FIGURE 5.2 – Les différentes étapes du décodage

EXERCICE #4 ▸ **Décodage**

1. Écrire **en pseudo-code** un algorithme qui réalise l'ajout à gauche d'un mot t de taille ℓ dans une matrice m dont les j premières colonnes sont remplies.
2. Écrire **en pseudo-code** un algorithme qui calcule la transformée inverse d'un mot quelconque. *Bien détailler l'analyse* ...
3. Quelle est la complexité de votre algorithme précédent ?