

Examen de Langage C

Durée : 1h30

Aucun document autorisé

Mobiles interdits Toutes les fonctions que vous écrirez doivent être clairement commentées avec des affirmations significatives (antécédents, conséquents, invariants). Vous prendrez soin de définir les bons paramètres et les bons types des données manipulées. Pensez à définir des fonctions auxiliaires si cela est nécessaire.

Question 1. Recherche (1/2)

Un tableau de taille n contient des entiers quelconques. On souhaite faire la somme uniquement des entiers appartenant $[a, b]$ (avec $a \leq b$).

- 1. Écrivez la fonction `somme` qui renvoie cette somme. Cette fonction admet 4 paramètres : le tableau, sa taille et les 2 bornes de l'intervalle.

```
/*
 * Antécédent : n>0 et t tableau initialisé,
 *              [a,b] intervalle d'entiers a<=b
 * Rôle : renvoie la somme des entiers pris sur l'intervalle
 *              [a,b] contenus dans le tableau t
 */
int somme(const int n, const int t[], const int a, const int b) {
    assert(n>0 && a<=b);
    int s=0;
    for (int i=0; i<n; i++)
        if (t[i]>=a && t[i]<=b)
            // t[i] ∈ [a;b]
            s+=t[i];
    // s = somme des t[i] ∈ [a;b]
    return s;
}
```

Question 2. Recherche (2/2)

La distance entre deux points a et b du plan cartésien de coordonnées (x_a, y_a) et (x_b, y_b) est égale à $\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$.

- 2. Écrivez la fonction `distance` qui prend en paramètre les coordonnées (réels `double`) de 2 points et renvoie la distance qui les sépare.

```
/*
 * Antécédent : (x_a, x_b) et (y_a, y_b) coordonnées
 *              de 2 points du plan cartésien
 * Rôle : renvoie la distance en les 2 points
 */
double distance(const double x_a, const double y_a,
```

```
const double x_b, const double y_b)
{
    return sqrt((x_b-x_a)*(x_b-x_a)+(y_b-y_a)*(y_b-y_a));
}
```

- 3. Un tableau tp de n (pair) réels représente les coordonnées de $n/2$ points du plan cartésien. Par exemple, le tableau tp qui contient les 10 réels $\{-1.9, 10.8, 0, 0, 0.5, 61.9, 0, 2.8, 110.9, 100\}$ représente 5 points de coordonnées $(-1.9, 10.8)$, $(0, 0)$, $(0.5, 61.9)$, $(0, 2.8)$ et $(110.9, 100)$.

Écrivez la fonction `distanceMin` qui renvoie la distance des deux points du plan qui sont les plus proches. Cette fonction a deux paramètres : le tableau tp de réels et n le nombre de points.

Vous appliquerez un algorithme *naïf* (pas très efficace, il y a mieux mais c'est plus cher 😊) qui consiste à tester toutes les paires de points. Pour n points, il y a $\frac{n \times (n-1)}{2}$ paires de points à tester. Par exemple, pour les 5 points précédents, la fonction renverra la distance minimale 2.8 (*i.e.* entre le deuxième et le quatrième point).

```
/*
 * Antécédent : tp tableau de n/2 points du plan
 *              n pair
 * Rôle : renvoie la distance minimale entre les points de tp
 */
double distanceMin(const double tp[], const int n) {
    assert((n&1)==0);
    // n est pair
    double distanceMin=DBL_MAX;
    // parcourir le tableau jusqu'à l'avant dernier point
    for (int pointCourant=0; pointCourant<n-3; pointCourant+=2) {
        // calculer la distance du point courant avec les points qui suivent
        for (int pointDistant=pointCourant+2; pointDistant<n-1; pointDistant+=2) {
            double newDistance = distance(tp[pointCourant], tp[pointCourant+1],
                                           tp[pointDistant], tp[pointDistant+1]);

            if (newDistance<distanceMin)
                // nouvelle distance min entre 2 points
                distanceMin=newDistance;
        }
    }
    return distanceMin;
}
```

Question 3. Nombres binaires

On représente un entier long non signé (`unsigned long int`) sous forme binaire à l'aide d'un tableau d'entiers courts (`short`) contenant des 0 et des 1. Le bit de poids faible (2^0) est à l'indice 0, et le bit de poids fort ($2^{\text{NB_BITS}-1}$) est à l'indice `NB\_BITS-1`. Par exemple, si `NB\_BITS` est égal à 8, le tableau qui contient `{0, 1, 1, 0, 1, 0, 0, 0}` représente l'entier décimal non signé 22. On définira le type `binaire`, comme suit :

```
typedef short binaire[NB_BITS];
```

- 4. Avec un `define`, définissez la constante `NB_BITS` égale au nombre d'éléments du tableau nécessaires à la représentation d'un entier long non signé indépendamment de sa représentation.

```
#define NB_BITS (sizeof(unsigned long int)*8)
```

-
- 5. Écrivez la fonction `binaireToLongInt` qui calcule et renvoie la valeur décimale d'un nombre binaire. Vous ne ferez aucune évaluation à la puissance. Cette fonction possède l'en-tête suivant :

```
unsigned int binaireToLongInt(const binaire b) {  
  
.....  
/*  
 * Antécédent : b nombre binaire  
 * Rôle : renvoie la valeur entière décimale non signée du  
 *         binaire b  
 */  
unsigned long int binaireToLongInt(const binaire b) {  
    // sauter tous les 0 à partir du bit de poids fort  
    int i=NB_BITS;  
    do i--; while (i>=0 && b[i]==0);  
    //  
    if (i<0)  
        // b ne contient que des 0  
        return 0;  
    // b[i] == 1  
    unsigned int n = 0;  
    for (; i>=0; i--)  
        n = (n<<1) + b[i];  
    // n valeur décimale de suite binaire contenue dans b  
    return n;  
}
```

.....

Question 4. Matrices

Une matrice carrée d'entiers $n \times n$ dont la demi-matrice supérieure ne contient que des 0 peut être représentée par un tableau à une dimension. Par exemple, les valeurs de la demi-matrice inférieure de la matrice 5×5 suivante :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 7 & 8 & 9 & 10 & 0 \end{pmatrix}$$

sont mémorisés dans le tableau à une dimension dans l'ordre suivant : [1 2 3 4 5 6 7 8 9 10]

- 6. Pour une matrice $n \times n$, quelle doit être la taille minimale du tableau à une dimension pour mémoriser les valeurs de la demi-matrice inférieure ?

$$\sum_{i=1}^n i = n(n+1)/2,$$

- 7. Écrivez la procédure `convertir` qui convertit la matrice $\text{mat } n \times n$ en le tableau à une dimension `demiMat` qui contiendra les entiers de la demi-matrice inférieure de `mat`. Cette procédure possède l'en-tête suivant :

```
void convertir(const int n, const int mat[][n], int demiMat[])
```

```
void convertir(const int n, const int mat[][n], int demiMat[]) {  
    int k=0;  
    for (int i=0; i<n; i++)  
        for (int j=0; j<=i; j++)
```

```
        demiMat[k++] = mat[i][j];  
    }
```

.....

- 8. Soit un élément `mat[i][j]` de la demi-matrice inférieure de la matrice carrée $n \times n$ `mat`, en fonction de i et j , à quel indice se trouve-t-il dans le tableau `demiMat` ?

.....

il est à l'indice $i*(i+1)/2+j$, c'est donc `demiMat[i*(i+1)/2+j]`. Cela correspond au nombre d'éléments de toutes les lignes qui précèdent la ligne i , plus l'indice j dans la ligne i .

.....

- 9. Écrivez la fonction `get` qui renvoie un élément de la matrice d'indice (i, j) dans le tableau `demiMat`. À l'aide de `assert`, vous vérifierez la validité des indices i et j . Cette fonction possède l'en-tête suivant :

```
int get(const int n, const int demiMat[], const int i, const int j) {
```

```
int get(const int n, const demiMat[], const int i, const int j) {  
    assert(i>=0 && i<n);  
    assert(j>=0 && j<n);  
    return j>i ? 0 : demiMat[i*(i+1)/2+j];  
}
```

.....

- 10. En utilisant la fonction `get` précédente, écrivez la procédure `print` qui, à partir d'un tableau `demiMat`, écrit sous forme matricielle ses éléments sur la sortie standard. Les 0 de la demi-matrice supérieure doivent donc apparaître. Cette procédure possède l'en-tête suivant :

```
void print(const int n, const int demiMat[])
```

```
/*  
 * Rôle : écrit sous forme matricielle la matrice carrée n*n  
 *         représentée par le tableau demiMat  
 */  
void print(const int n, const int demiMat[]) {  
    for (int i=0; i<n; i++) {  
        for (int j=0; j<n; j++)  
            printf("%3d", get(n, demiMat, i, j));  
        printf("\n");  
    }  
}
```

.....