TD 1 – Tableaux

Vecteurs

1) Initialisation d'un tableau

a) Écrire une fonction prenant en argument un tableau de type `array[1..N] of integer`, et le remplissant de zéros.

b) Écrire une fonction prenant en argument un tableau `t` de type `array[1..N] of integer`, et le remplissant avec les valeurs d'une fonction `f` déclarée ailleurs dans le code. La case `t[i]` doit contenir la valeur `f(i)`.

c) Écrire une fonction prenant en argument un tableau de type `array[1..N] of integer`, et le remplissant de valeurs aléatoires.

Indication : on rappelle que `random(m)` renvoie un entier pseudo-aléatoire compris entre 0 et m.

2) Impression d'un tableau

a) Écrire une fonction prenant en argument un tableau de type `array[1..N] of integer`, et l'affichant sur la console. Le tableau devra apparaître sous la forme

```
[ 1; 2; 5; 3; 7; 18; 1]
```

b) Modifier cette fonction de manière à ce qu'elle n'imprime que les 20 premiers éléments du tableau, suivis de points de suspension si `N` est plus grand que 20.

3) Moyennes

a) Écrire une fonction `moyenne` prenant en argument un tableau de réels, et renvoyant la moyenne de ses éléments.

b) Écrire une fonction `moyenne_elaguee` prenant en argument un tableau de réels, et renvoyant de même la moyenne de ses éléments, *élaguée du plus petit et du plus grand élément du tableau*.

Par exemple, la moyenne élaguée du tableau `[12.0; 14.5; 8.1; 15.3]` est `13.25=(12+14.5)/2`, car `8.1` et `15.3` sont enlevés de la série.

4) Appliquer une fonction aux éléments d'un tableau

a) Écrire une fonction prenant en argument un tableau d'entier `t` de type `array[1..N] of integer`, et augmentant tous ses éléments de 1.

b) Modifier votre fonction de manière à ce qu'elle remplace chaque élément de `t` par son image par la fonction `f` définie autre part dans le code.

5) Ce tableau est-il trié ?

a) Écrire une fonction `croissant` prenant en argument un tableau de type `array[1..N] of integer`, et renvoyant vrai si et seulement si le tableau est trié en ordre croissant.

b) Modifier cette fonction de manière à ce qu'elle renvoie 0 si le tableau n'est pas trié, 1 si le tableau est trié en ordre croissant, et -1 s'il est trié en ordre décroissant.

Attention au fait que les éléments du tableau ne sont pas nécessairement tous distincts.

6) Miroir d'un tableau

a) Écrire une fonction `miroir` prenant en argument un tableau de type `array[1..N] of integer`, et en renvoyant l'*image miroir*, en échangeant `t[1]` et `t[N]`, `t[2]` avec `t[N-1]`, etc.

b) Modifier votre fonction de manière à ce qu'elle prenne deux arguments supplémentaires `u` et `v`, et qu'elle retourne le tableau `t` après avoir retourné la partie constituée des case d'indices compris entre `u` et `v` (inclus). Cette fonction sera utilisée plus bas.

7) Avant de trier, on mélange, et on se fait un petit poker !

On souhaite réaliser une procédure qui mélange aléatoirement les éléments d'un tableau donné en paramètre (par exemple pour simuler le mélange d'un jeu de carte).

a) Écrire une procédure prenant en argument un tableau d'entiers de type `array[1..N] of integer` et deux entiers `i` et `j`, et échangeant les éléments d'indices `i` et `j` du tableau.

En déduire une première procédure de mélange.

b) On souhaite simuler la méthode de mélange qu'on voit souvent dans les films : on coupe le jeu en deux, et on insère les cartes d'un des deux paquets entre les cartes de l'autre.

Pour cela, on va composer un nouveau tableau, et le remplir en prenant aléatoirement des éléments de la première ou de la seconde partie du tableau initial. Écrire la procédure mettant en oeuvre cette idée.

c) Utiliser ces deux méthodes de mélange pour simuler un grand nombre de distribution de cinq cartes d'un jeu de 52 cartes, et calculant la fréquence des figures usuelles du poker : une paire, deux paires, une couleur...

8) Recherche dans un tableau

a) Écrire une fonction prenant en argument un tableau d'entiers `t`, de type `array[1..N] of integer`, et un entier `x`, et renvoyant le nombre d'occurrences de `x` dans `t`.

b) Écrire une fonction prenant les mêmes arguments, et renvoyant `0` si `x` ne se trouve pas dans `t`, et l'indice de la première position (on dit *occurrence*) de `x` dans `t` dans le cas contraire.

9) Extrema d'un tableau

a) Écrire une fonction prenant en argument un tableau d'entiers `t`, de type `array[1..N] of integer`, et renvoyant le plus grand élément présent dans `t`.

b) Écrire une fonction prenant les mêmes arguments, et renvoyant le plus grand et le plus petit élément du tableau `t`.

Matrices

1) Initialisation d'un tableau à deux dimensions

a) Écrire une fonction prenant en argument une matrice de type `array[1..N,1..M] of integer`, et initialisant l'ensemble de ses cases à `0`.

b) Modifier cette fonction pour qu'elle remplisse le tableau d'entiers positifs aléatoires.

2) Moyenne des éléments d'une matrice

Écrire une fonction calculant la moyenne des termes d'une matrice de type `array[1..N,1..M] of real`.

3) Opérations matricielles

a) Écrire une fonction prenant en argument deux matrices A et B de type `array[1..N,1..M] of integer`, et calculant leur *somme* $C = A + B$.

b) Écrire une fonction prenant en argument deux matrices *carrées* A et B de type `array[1..N,1..N] of integer`, et calculant leur *produit* $C = A \times B$.

c) Reprendre la question précédente avec des matrices rectangulaires A et B, dont le produit $A \times B$ est calculable. Voyez-vous un problème éventuel ? Comment vous proposez-vous de le résoudre ?

4) Saisie et impression d'une matrice

a) Écrire une fonction prenant en argument une matrice `A` de type `array[1..N,1..N] of integer`, deux entiers `p` et `q`, qui écrit sur l'écran la partie de `A` dont les lignes sont comprises entre `1` et `p`, et dont les colonnes sont comprises entre `1` et `q`. On supposera dans un premier temps que les entiers stockés dans la matrice n'ont pas plus de 3 chiffres (mais ils peuvent être négatifs !).

b) Écrire une fonction demandant à l'utilisateur deux entiers `p` et `q`, et les éléments d'une matrice `A` de taille $p \times q$. On stockera le tout dans un tableau d'entiers de type `array[1..N,1..N] of integer`.

Des exercices plus compliqués

1) Rotation d'un tableau

On veut écrire une fonction `rotation` prenant en argument un tableau de type `array[1..N] of integer`, et un entier k, et renvoyant la *rotation* d'ordre k de ce tableau : `t[1]` va en `t[k+1]`, `t[2]` et `t[k+2]`,... `t[N-k-1]` en `t[N]`, `t[N-k]` en `t[1]`,... et `t[N]` en `t[k]`.

a) Une première idée consiste à utiliser un tableau auxiliaire pour recopier le premier tableau, avant de faire une boucle pour remplir le tableau à modifier. Implémenter cette idée.

b) Une deuxième idée, plus difficile à mettre en oeuvre, consiste à sauvegarder `t[1]`, placer `t[k+1]` dans `t[1]`, puis `t[2k+1]` dans `t[k+1]`, et ainsi de suite.
Si tout se passe bien (quand est-ce le cas ?), l'algorithme se termine en une seule passe. Dans le cas contraire, il faut poursuivre par la première case qui n'a pas encore été modifiée, et ainsi de suite.
Implémenter cette idée.

c) Une idée beaucoup plus simple, mais plus astucieuse, consiste à constater qu'on peut se ramener à des images miroirs. En effet, si la structure du tableau est |A|B|, on peut obtenir le résultat final |B|A| en prenant le miroir de la partie A, puis le miroir de la partie B, et enfin le miroir du tableau obtenu. Faire un petit dessin pour vérifier ceci, et implémenter cette idée à l'aide de la fonction `miroir` construite plus haut.

2) Recherche dans un tableau, version ultra-rapide

Lorsque le tableau dans lequel on effectue une recherche est trié (en ordre croissant), on peut accélérer grandement la recherche en utilisant la méthode de *dichotomie*. Elle s'appuie sur le pseudo-algorithme suivant :

- au départ, l'élément `x` cherché doit se trouver dans le tableau `t[1..N]` ; on initialise `u` à `1` et `v` à `N` ;
- (B) si `u>v`, alors `x` ne se trouve pas dans le tableau ; on retourne `0` ou `false`, selon ce qui nous intéresse ;
  sinon, soit `m=(u+v)/2` ; on compare `x` à `t[m]` :
  - si `x=t[m]`, on retourne `true`, ou la valeur de `m`
  - si `x<t[m]`, `x` ne peut se trouver que dans la partie `t[u,m-1]` du tableau ; on pose `v=m-1`, et on retourne en (B) ;
  - sinon (i.e. si `x>t[m]`), `x` ne peut se trouver que dans la partie `t[m+1,v]` du tableau ; on pose `u=m+1`, et on retourne en (B).

a) Écrire un algorithme précis implémentant cette méthode.

b) L'implémenter en Pascal, et vérifier qu'il fonctionne sur un échantillon suffisamment large de cas (tester la recherche d'un élément présent, mais aussi d'un élément absent, plus grand, plus petit ou entre les valeurs extrêmes du tableau).

c) Dans une version encore plus sophistiquée de la recherche dichotomique, on cherche à mieux estimer la partie du tableau dans laquelle pourrait se trouver l'élément cherché.
Si par exemple on recherche l'élément `12` dans un tableau (ou une partie de tableau) dont les valeurs extrêmes sont `1` et `10000`, on sent bien qu'on va perdre son temps à aller explorer le milieu du tableau si les valeurs de ses éléments sont à peu près équitablement réparties. On préférera tester une position plus proche de `1`, et si possible à peu près proportionnelle à l'écart entre `1` et `10000`.
Proposer un algorithme remplaçant le calcul du milieu `m` par une position plus intéressante, et discuter des possibles problèmes.
Implémenter cet algorithme en Pascal, et le tester sur un exemple suffisamment grand pour pouvoir se faire une idée (on pourra par exemple générer un tableau d'un millier de cases, tel que la case n°k contienne un entier au hasard entre $10k$ et $10\,(k+1)$).

3) Crible d'Erathostene

Le principe du crible d'Erathostene est le suivant : une fonction prend en argument un entier `N`, et renvoie un tableau `Prem[2..N]` de booléens tel que `Prem[i]` est vrai si et seulement si `i` est premier (c'est-à-dire divisible uniquement par 1 et par lui-même ; on rappelle que `1` n'est pas premier !). Pour cela :

- on initialise chaque case du tableau `Prem` à `Vrai`, et l'entier `k` à `2` ;
- tant que `k` est inférieur à la racine carrée de `N`,
  - on met à `Faux` chaque case `Prem[k*i]`, pour `i` plus grand que `2` (c'est l'étape de *crible*)
  - on remplace `k` par la valeur du premier indice `k'` plus grand que `k` tel que `Prem[k']` soit `Vrai`.

a) Implémenter cet algorithme en Pascal.

b) Utiliser la fonction `crible` ainsi créée pour écrire une fonction `Pi` qui prend en argument un entier `N` et renvoie le nombre de nombres premiers inférieurs ou égaux à `N`.

c) Utiliser la fonction `crible` pour afficher tous les couples de `nombres premiers jumeaux`, c'est-à-dire des couples de nombre premiers (p, q) tels que $q = p + 2$ (on pense qu'il existe une infinité de tels couples, sans avoir pour l'instant réussi à le démontrer).

4) Recherche de caractéristiques de tableaux

a) Recherche de motif

On voudrait écrire une fonction prenant en argument un tableau `t[1..N]` d'entiers, et un autre tableau, `m[1..k]` d'entiers, et renvoyant l'indice de la première *occurrence* du tableau `m` dans `t`.

Plus précisément, cette fonction devra renvoyer l'entier `n` si[1] `t[n]=m[1]`, `t[n+1]=m[2]`,... et `t[n+k-1]=m[k]`, et si l'entier est le plus petit entier pour lequel ceci est réalisé. La fonction devra renvoyer `0` si cette occurrence n'est pas trouvée.

i. Imaginer un algorithme mettant en œuvre cette recherche, en "superposant" le tableau `m` au tableau `t` dans toutes les positions possibles.

ii. Implémenter cet algorithme en Pascal.

iii. Combien de comparaisons suffisent dans le meilleur des cas pour obtenir l'indice cherché ? Quand cela se produit-il ?

iv. Combien de comparaisons faut-il faire au pire pour terminer le calcul ? Donner un exemple simple dans lequel ce pire cas est réalisé.

v. Chercher sur Internet des références sur le sujet de la recherche de motifs, et trouver un ou deux algorithmes sensiblement plus performants que l'algorithme "naïf" développé ici (les noms de KNUTH et MOORE devraient apparaître au fil de vos recherches !).

b) Recherche d'une plus longue suite constante

i. Écrire une fonction prenant en argument un tableau `t[1..N]` d'entiers et un entier `n`, et renvoyant la longueur de la plus longue suite *consécutive* de cases du tableau `t` égales à `n`.
Ainsi, votre fonction doit renvoyer `k` s'il existe un indice `i` tel que

```
t[i]=t[i+1]=...=t[i+k-1]=n
```

et si il n'y a pas de séquence plus longue présente dans le tableau.

ii. Indiquer le nombre de comparaisons nécessaire pour terminer le calcul.

c) Recherche d'une plus longue suite strictement croissante

i. Écrire une fonction prenant en argument un tableau `t[1..N]` d'entiers , et renvoyant la longueur de la plus longue suite *consécutive* de cases du tableau `t` rangées en ordre strictement croissant.
Ainsi, votre fonction doit renvoyer `k` s'il existe un indice `i` tel que

```
t[i]<t[i+1]<...<t[i+k-1]
```

et si il n'y a pas de séquence plus longue présente dans le tableau.

ii. Indiquer le nombre de comparaisons nécessaire pour terminer le calcul.

[1] Une autre manière d'exprimer cela est de dire que `m` est un *sous-tableau* du tableau `t`.