Olympiades d'Informatique

http://uclouvain.acm-sc.be/olympiades

Exemples de questions pour le supérieur

Ce document propose des exemples de questions pour le concours destiné aux élèves du supérieur. La première section donne des exemples de questions de logique tandis que la seconde propose des exemples de questions algorithmiques. Nous vous conseillons de lire le document *« Introduction à l'algorithmique »* pour comprendre les exemples de solution donnés. Pour des exemples de questions de logique, consultez le document *« Exemples de questions pour les secondaires »*.

1 Algorithmique

1.1 Recherche d'une sous-chaine

> Écrivez un programme qui lit deux chaines de caractères sur l'entrée standard. La première chaine W correspond à un mot constitué de lettres (`a-zA-Z`) et la seconde chaine P à un motif constitué de lettres (`a-zA-Z`) et du symbole `_`. Le programme doit tester si la chaine P est une sous-chaine de la chaine W sachant que `_` remplace n'importe quelle lettre.

Une solution à ce problème est assez simple à obtenir avec une double boucle. La première boucle va parcourir la chaine W pour tester les différentes positions possibles où le motif pourrait se trouver. La seconde boucle permet de tester les différents caractères du motif. Il faut faire attention au cas où la chaine P est plus longue que la chaine W et renvoyer **false** dans ce cas. Ceci est géré par la seconde partie de la condition de la boucle `while`.

Algorithme 1 : Recherche d'une sous-chaine.

Input : W, un tableau de caractères appartenant à (`a-zA-Z`), de longueur $lenW > 0$ et P un tableau de caractères appartenant à (`a-zA-Z_`), de longueur $lenP$

Output : **true** si la W est un sous-tableau de P, sachant que `_` remplace n'importe quel caractère, et **false** sinon

```
// On teste chaque position possible dans W
found ← false
pos ← 0
while not found and pos ≤ lenW − lenP do
    // On teste si P se trouve en position pos
    match ← true
    i ← 0
    while match and i < lenP do
        if P[i] ≠ '_' and P[i] ≠ W[pos + i] then
            match ← false
        i ← i + 1
    // Si on a trouvé P dans W, on peut arrêter
    if match then
        found ← true
    pos ← pos + 1
return found
```

1.2 Tri de tableau

> Vous disposez de deux fonctions `moveElem` et `minIndex` dont on vous fournit les spécifications. On vous demande d'écrire un algorithme qui permet de trier un tableau de taille donnée n de manière croissante, **en n'utilisant que ces deux fonctions** sur le tableau.
> La fonction `moveElem (tab, i, j)` permet d'intervertir les éléments d'indices `i` et `j` du tableau `tab`, `i` et `j` étant deux indices valides de `tab`. La fonction `minIndex (tab, i)` renvoie l'indice d'un plus petit élément du sous-tableau `tab[i:n-1]`, c'est-à-dire un indice j tel que $\forall k \in [i, n-1] : tab[k] \geq tab[j]$, l'indice `i` étant un indice valide de `tab`.

Pour cette question, il faut d'abord bien comprendre ce que font les fonctions qu'on peut utiliser. Une fois celles-ci comprises, on va facilement pouvoir écrire un algorithme qui va trier les éléments d'un tableau de manière croissante. Voyons comment on va faire à partir du schéma suivant :

	`0`	`i` `n-1`
`tab`	A	B

Le tableau `tab` est divisé en deux parties A (des indices 0 à $i-1$) et B (des indices i à $n-1$). Tous les éléments se trouvant dans A sont plus petits que ou égaux à tous ceux se trouvant dans B. De plus, le sous-tableau A est trié de manière croissante :

$$\forall a \in A : \forall b \in B : a \leq b \qquad\qquad \forall i \in [0, i-2] : tab[i] \leq tab[i+1]$$

L'algorithme va procéder en trois étapes :

1. On retrouve un indice d'un élément minimal dans B (c'est-à-dire un indice j tel que $\forall i \in [i, n-1] : tab[j] \leq tab[i]$) en utilisant `minIndex` ;
2. On déplace ensuite l'élément d'indice j pour le placer à l'indice i en utilisant `moveElem` ;
3. On incrémente la valeur de i de un et on boucle tant que $i < n-1$.

L'algorithme qui résout le problème prend donc la forme suivante :

Algorithme 2 : Tri d'un tableau.

Input : *tab*, un tableau de longueur $n > 0$
Output : Un tableau contenant les mêmes éléments que *tab*, mais trié de manière croissante

$i \leftarrow 0$
while $i < n-1$ **do**
  $j \leftarrow$ `minIndex`(tab, i)
  `moveElem`(tab, i, j)
  $i \leftarrow i + 1$

return *tab*

On n'est bien entendu pas obligé de renvoyer la valeur de *tab*. En effet, l'algorithme pourrait simplement se limiter à modifier les valeurs du tableau qui lui est donné.

1.3 Comparaison du premier et dernier élément d'une liste

Contexte : On peut voir une liste comme une paire de deux éléments : la tête (`H`) et la queue (`T`). On peut dès lors représenter une liste comme `L = H|T`. Les seules opérations permises sur une liste `L` sont `head(L)` qui permet de récupérer la tête et `queue(L)` qui permet d'obtenir la queue. La liste vide est représentée par `nil`.

Exemple : Soit la liste `L = 1|2|3|4|5`. La tête de la liste est `head(L) = 1` et sa queue est `queue(L) = 2|3|4|5`. La tête de la queue est `head(queue(L)) = 2` et sa queue est `queue(queue(L)) = 3|4|5`, etc.

Énoncé : On vous demande d'écrire un algorithme qui permet de tester si le premier et le dernier élément d'une liste `L` non-vide de longueur n sont identiques ou non. Vous ne pouvez utiliser les opérations `head` et `queue` qu'au maximum n fois.

Ici, on doit pouvoir retrouver le premier et le dernier élément d'une liste afin de les comparer. Les seules opérations permises sur la liste sont `head` qui permet de récupérer le premier élément et `queue` qui permet de récupérer le reste de la liste (la liste sans sa tête). On sait également qu'une liste vide est représentée par `nil`. Enfin, on ne peut utiliser `head` et `queue` qu'au maximum n fois, n étant la longueur de la liste.

Algorithme 3 : Comparaison du premier et dernier élément d'une liste.

Input : L, une liste non-vide
Output : **true** si le premier et dernier élément de L sont identique et **false** sinon

$first \leftarrow$ `head`(L)
$last \leftarrow first$
$tmp \leftarrow$ `queue`(L)
while $tmp \neq$ `nil` **do**
  $last \leftarrow$ `head`(tmp)
  $tmp \leftarrow$ `queue`(tmp)
return $first = last$

1.4 Labyrinthe

Soit un labyrinthe pouvant être représenté par un ensemble de $M \times N$ cases. On peut se déplacer dans le labyrinthe d'une case à une autre si elles partagent un coté (pas de déplacement diagonal). L'algorithme reçoit les valeurs M et N, ainsi qu'un tableau de taille $M \times N$ dont une case $tab[i][j]$ vaut 0 si la case est vide et qu'on peut passer, 1 si la case est bloquée, 2 si c'est la case de départ et 3 si c'est la case d'arrivée. L'algorithme doit renvoyer 0 s'il n'existe pas de chemin entre le départ et l'arrivée et 1 sinon.

(*Aide :* une file pourrait être utile)

1.5 Recherche de pattern

Soient deux listes d'entiers `List` et `Sublist`, non-vides. Écrivez un algorithme qui renvoie le plus petit indice i de `List`, tel que la sous-liste `List[i, i + len (Sublist)[` soit égale à `Sublist`. Si un tel indice n'existe pas, l'algorithme renvoie -1.

1.6 Addition de nombres binaires

Soient deux listes `List1` et `List2` d'entiers 0 et 1. Écrivez un algorithme qui calcule la somme des deux nombres binaires représentés par ces listes (le bit le moins significatif se trouvant à l'indice 0 des listes).

1.7 Fibonacci

La suite de Fibonacci est une suite d'entiers qui commence ainsi : $0, 1, 1, 2, 3, 5, 8, \cdots$ Les deux premiers termes de la suite sont donc 0 et 1. Chacun des termes suivants est égal à la somme des deux termes précédents. Ainsi, si on note par Fib_n, le n^e terme de la suite, on a $Fib_1 = 0$, $Fib_2 = 1$ et $Fib_{i+2} = Fib_{i+1} + Fib_i$ pour $i > 2$. Écrivez un algorithme qui calcule le n^e terme de la suite de Fibonnaci, pour $n > 0$.

1.8 Triangle de Pascal

Le triangle de Pascal est un triangle de nombres qui se construit comme suit : les deux bords descendants du triangle sont remplis de 1. Ensuite, chaque case du triangle est égal à la somme des deux cases se trouvant au-dessus d'elle. Voici le début de ce triangle :

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Écrivez un algorithme qui prend deux paramètres L et C et qui calcule la valeur qui doit se trouver dans la C^e case de la L^e ligne du triangle de Pascal. Si une telle case n'existe pas, l'algorithme doit renvoyer -1.

1.9 La plus longue sous-chaine

Soient deux chaines A et B. Écrivez un algorithme qui recherche la plus longue sous-chaine de B qui est également une sous-chaine de A. L'algorithme renvoie trois valeurs : i est l'indice de la sous-chaine dans A, j l'indice de la première occurrence de la sous-chaine dans B et ℓ est la longueur de la sous-chaine.

Par exemple, pour les chaines $A = rattrapage$ et $B = trappe$, l'algorithme renvoie $\langle 3, 0, 4 \rangle$, ce qui correspond à la sous-chaine *trap*. Pour les chaines $A = ta$ et $B = ratata$, l'algorithme renvoie $\langle 0, 2, 2 \rangle$. Et donc, si la chaine B ne contient pas de sous-chaine qui est sous-chaine de A, l'algorithme doit renvoyer $\langle x, 0, 0 \rangle$, où x peut valoir n'importe quoi entre 0 et $n - 1$ avec n la longueur de la chaine A.

2 QCM

2.1 Puissance

Soient les deux algorithmes `pow_1` et `pow_2` suivants :

Algorithme 4 : `pow_1`

Input : X un nombre réel et N un naturel
Output : La valeur de X^N

if $N = 0$ **then**
  $result \leftarrow 1$
else
  $result \leftarrow X * \texttt{pow\_1}(X, N - 1)$
return *result*

Algorithme 5 : `pow_2`

Input : X un nombre réel et N un naturel
Output : La valeur de X^N

if $N = 0$ **then**
  $result \leftarrow 1$
else if $N \bmod 2 = 1$ **then**
  $result \leftarrow X * \texttt{pow\_2}(X, N - 1)$
else
  $Y \leftarrow \texttt{pow\_2}(X, N/2)$
  $result \leftarrow Y * Y$
return *result*

1. Pour les mêmes données d'entrée (X et N) et assez grandes, lequel des deux algorithmes s'exécutera en faisant le moins de calculs (le moins de récursion) ?
2. Si on multiplie par deux la valeur de X, combien de récursions seront ajoutées avec l'algorithme `pow_1` ?
3. Si on multiplie par deux la valeur de N, combien de récursions seront ajoutées avec l'algorithme `pow_2` ?

2.2 Liste de diviseurs

On souhaite écrire une fonction qui prend en entrée un tableau *tab* d'entiers strictement positifs et qui renvoie un tableau d'entiers de longueur 5, où $retour[i]$ contient le plus petit entier du tableau *tab* qui est divisible par i, pour $i > 0$, et $retour[0]$ contient 0. Voici le squelette de la fonction en Java :

```java
int[] trierParDivisibilite (int[] tableau)
{
   int retour[] = new int[5];
   // à compléter
}
```

Comment compléter la fonction ?

(a)
```java
for (int i = 1; i < 5; i++) {
   if (tableau[i] % i == 0 && retour[i] > tableau[i]) {
      retour[i] = tableau[i];
   }
}
```

(b)
```java
for (int x : tableau) {
   for (int i = 1; i < 5; i++) {
      if (x % i == 0 && (retour[i] > x || retour[i] == 0)) {
         retour[i] = x;
      }
   }
}
```

(c)
```java
for (int x : tableau) {
   for (int i = 1; i < 5; i++) {
      if (retour[i] % i && x > retour[i]) {
         retour[i] = tableau[i];
      }
   }
}
```