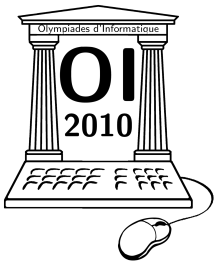


Olympiades d'Informatique

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.

1 Logique

2 Algorithmique

2.1 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.

Algorithme 1 : Recherche d'une sous-chaine.

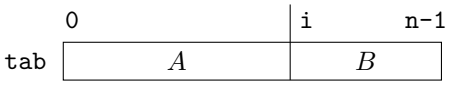
Input : W, un tableau de caractères appartenant à (`a-zA-Z`), de longueur $n > 0$ et P un tableau de caractères appartenant à (`a-zA-Z_`), de longueur $m \leq n$
Output : **true** si la W est un sous-tableau de P, sachant que `_` remplace n'importe quel caractère, et **false** sinon

```
1  // On teste chaque position possible dans W
2  found ← false
3  pos ← 0
4  while not found and pos ≤ n − m do
5      // On teste si P se trouve en position pos
6      match ← true
7      i ← 0
8      while match and i < m do
9          if P[i] ≠ W[pos + i] then
10             match ← false
11         m ← m + 1
12     // Si on a trouvé P dans W, on peut arrêter
13     if match = true then
14         found ← true
15 return found
```

2.2 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 :



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 ou égaux à tout ceux se trouvant en B. De plus, le sous-tableau A est trié de manière croissante :

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

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

```
1 i ← 0
2 while i < n − 1 do
3     j ← minIndex(tab, i)
4     moveElem(tab, i, j)
5 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.

2.3 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

```
1 first ← head(L)
2 last ← first
3 tmp ← queue(L)
4 while tmp ≠ nil do
5 |   last ← head(tmp)
6 |   tmp ← queue(tmp)
7 return first = last
```