400 000
(MIMP - SP1)

USTL - LST-A 2007-2008

Documents autorisés

Examen API2

juin 2008

Durée : 2h00

Exercice 1. *Séparation d'une liste en deux listes*

On souhaite séparer une liste l d'entiers naturels en deux listes : l_1 constituée des entiers pairs de l, et l_2 constituée des entiers impairs.

Pour les deux programmes demandés dans cet exercice, vous n'utiliserez que les opérations primitives sur les listes telles qu'elles ont été décrites en cours.

Question 1.1. Dans cette question, on souhaite que la liste initiale ne soit pas modifiée, et par conséquent que les deux listes construites soient obtenues en recopiant les entiers de la liste de départ.

Q 1.1–1. Proposez-vous une fonction ou une procédure pour réaliser cette opération ? Donnez-en l'entête.

Q 1.1–2. Réalisez cette fonction ou procédure.

Q 1.1–3. Indiquez très précisément les listes l_1 et l_2 obtenues lorsque votre programme est appliqué à la liste $l = (1, 6, 1, 8, 0, 3)$.

Question 1.2. On veut maintenant que la construction des deux listes se fasse sans aucune allocation de mémoire.

Réalisez la procédure dont l'entête est

```
procedure separer(var l1 : LISTE; out l2 : LISTE);
```

Après l'instruction `separer(`l_1`,`l_2`);`, la liste l_1 est la liste des éléments pairs, et la liste l_2 celle des éléments impairs de la liste initiale l_1.

Exercice 2. *Poids maximum*

On considère une suite S de n entiers relatifs ($n > 0$) :

$$S = S_1, S_2, \ldots, S_n$$

On appelle ***segment*** de S, et on note (i, j), un couple d'indices i et j tels que $1 \leqslant i \leqslant j \leqslant n$. Nous disons que i et j sont les extrêmités gauche et droite du segment (i, j).

Un segment (i, j) de S caractérise la sous-suite $S_{i..j}$ d'éléments adjacents de S. Le ***poids*** d'un segment est la somme des éléments de la sous-suite qu'il caractérise.

On étudie dans cet exercice un algorithme de calcul du ***poids maximum*** des segments de S.

Exemple : Soit la suite $S = 43, -52, 78, 15, -67, 39, 93, -99, -32, 48$.
Le segment $(2, 4)$ caractérise la sous-suite $-52, 78, 15$ et son poids est $-52 + 78 + 15 = 41$.
Le poids maximum de la suite S est 158 ; c'est le poids du segment $(3, 7)$.

On suppose que la suite est stockée dans un tableau d'entiers relatifs (positifs et négatifs) `S[1..n]`, avec `1=low(S)` et `n=high(S)`.

Question 2.1. Écrivez une fonction récursive nommée **poids_rec**, paramétrée par le tableau `S` et deux entiers `i` et `j`, et qui retourne le poids du segment (i, j).

Question 2.2. Quelle est, en fonction de i et j, la complexité de cette fonction en nombre d'additions d'éléments de `S` ?

Question 2.3. Donnez une version itérative, nommée **poids_iter**, de la fonction **poids_rec**.

Question 2.4. Quel est le nombre d'additions d'éléments de `S` que le calcul de `poids_iter(S,i,j)` engendre ?

Pour calculer le poids maximum des segments de `S`, on s'appuie sur le principe suivant : tout en énumérant tous les segments possibles de la suite S, on calcule le poids de chaque segment, et on retient le poids maximum des segments déjà rencontrés.

Pour décrire l'énumération, on répartit l'ensemble des segments selon la valeur de l'extrémité droite. Les segments d'extrémité droite égale à j (j variant de 1 à n) sont énumérés en faisant varier l'extrémité droite i de 1 à j

j	liste des segments
1	$(1,1)$
2	$(1,1),\ (1,2)$
3	$(1,1),\ (1,2),\ (1,3)$
$\vdots$	$\vdots$
n	$(1,1),\ (1,2),\ (1,3),\ \ldots,\ (1,n)$

Question 2.5. Écrivez une fonction nommée `poids_max`, paramétrée par le tableau `S` et qui retourne le poids maximum de `S`. Vous pourrez utiliser la fonction `max` qui renvoie le plus grand des deux entiers passés en paramètre.

Question 2.6. Quel est le nombre d'additions d'éléments de `S` que cette fonction engendre ? On rappelle les identités suivantes :

$$1+2+\cdots+k \;=\; \frac{k(k+1)}{2} \qquad \text{et} \qquad 12+22+\cdots+k2 \;=\; \frac{k(k+1)(2k+1)}{6}$$

On note $\texttt{pmd}(S,i)$ le poids maximum des segments de S dont l'extrémité droite est i, c'est-à-dire le poids maximum des segments $(1,\,i),(2,\,i),\cdots,(i,\,i)$. $\texttt{pmd}(S,i)$ peut être défini par récurrence comme suit :
- $\texttt{pmd}(S,1) = S_1$
- pour $i > 1$, $\texttt{pmd}(S,i)$ doit être défini en termes de $\texttt{pmd}(S,i-1)$ et de S_i.

Chaque segment d'extrémité droite égale à $i-1$ permet de construire un segment d'extrémité droite égale à i, simplement en ajoutant l'élément supplémentaire S_i. Tous les segments d'extrémité droite égale à i sont construits de cette manière sauf le segment $(i,\,i)$. Le segment de poids maximum se terminant en i est soit
- le segment $(i,\,i)$ dont le poids est S_i
- un segment $(k,\,i)$ tel que $1 \leqslant k < i$. Son poids est $\textbf{poids}(S,k,i-1) + S_i$. Pour que ce poids soit maximum, il est nécessaire que $\textbf{poids}(S,k,i-1) = \texttt{pmd}(S,i-1)$.

$$\begin{cases} \texttt{pmd}(S,1) &= S_1 \\ \texttt{pmd}(S,i) &= \max(\texttt{pmd}(S,i-1)+S_i,\ S_i) \end{cases}$$

Question 2.7. Déduisez7 de ces conditions un nouvel algorithme récursif de calcul du poids maximum des segments de la suite S.

Question 2.8. Quel est le nombre d'additions d'éléments de S que cet algorithme engendre ?

Exercice 3. *Expression d'arbres*

Le but de cet exercice est de réaliser une procédure qui, à partir d'un arbre binaire passé en paramètre, écrit l'expression décrivant cet arbre à l'aide de l'opération primitive `creerArbre` et de la constante `ARBREVIDE`. Le type des éléments contenus dans chaque nœud est supposé être le type `CARDINAL`.

Par exemple avec l'arbre de la figure 1, la procédure doit écrire l'expression montrée sur la figure 2.

Question 3.1. Réalisez la procédure

```
procedure exprimerArbre(const a : ARBRE);
```

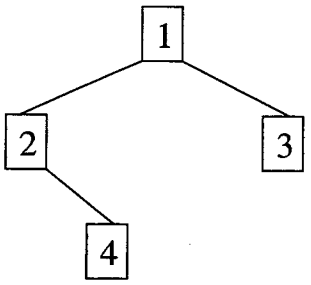

FIG. 1 – L'arbre à exprimer

```
creerArbre(1,
           creerArbre(2,
                      ARBREVIDE,
                      creerArbre(4,
                                 ARBREVIDE,
                                 ARBREVIDE)),
           creerArbre(3,
                      ARBREVIDE,
                      ARBREVIDE))
```

FIG. 2 – L'expression décrivant l'arbre de la figure 1

qui, contrairement à ce qui est montré sur la figure 2, produit l'écriture d'une telle expression sur une seule ligne.

Question 3.2. Réalisez une seconde version de cette procédure qui formate l'écriture de l'expression comme elle est montrée sur la figure 2, c'est-à-dire avec un passage à la ligne et une indentation faisant en sorte que les trois paramètres de chaque appel à `creerArbre` soient superposés.

Cette procédure comprendra un second paramètre donnant l'indentation de chaque ligne.

40000
MIMP - SP1

USTL - LST-A 2007-2008 Algorithmique et Programmation impérative

Documents autorisés

Examen API2

janvier 2008

Durée : 2h00

Exercice 1.

On suppose que l'on dispose de deux listes chaînées triées L_1 et L_2 de longueur n_1 et n_2 respectivement.

Question 1.1. *Écrire une fonction récursive qui fusionne les deux listes L_1 et L_2 pour produire une seule liste triée formée des éléments des deux listes initiales. On peut supposer que les éléments des deux listes sont tous distincts.*

Question 1.2. *Quelle est la complexité de cette fonction en nombre d'appels à la fonction primitive `ajouteEnTete` en fonction des longueurs n_1 et n_2 des listes L_1 et L_2 ?*

On suppose maintenant que l'on dispose de k listes chaînées triées $L_1, L_2, \ldots, L_k$ ayant chacune p éléments. On souhaite fusionner toutes ces listes en une seule liste triée de $n = p \times k$ éléments. On propose l'algorithme suivant :

- on fusionne les deux premières listes
- on fusionne la liste obtenue à l'étape précédente avec la troisième liste
- on fusionne la liste obtenue à l'étape précédente avec la quatrième liste
- et ainsi de suite

Question 1.3. *Analyser le temps de calcul de cet algorithme et calculer sa complexité en nombre d'appels à la fonction `ajouteEnTete`.*

Exercice 2.

On suppose que l'on dispose des prix d'un article pour n jours consécutifs, prix notés $p_0, p_1, \ldots, p_{n-1}$ (p_j désigne le prix de l'article pour le jour j).

On souhaite, pour chaque jour j, calculer le nombre maximum de jours consécutifs qui précèdent j et dont les prix sont inférieurs ou égaux à celui de j. On notera ce nombre s_j. Formellement, s_j est égal au plus grand entier k tel que

$$k \leq j+1 \text{ et } p_i \leq p_j \text{ pour } i = j-k+1, \ldots, j$$

On considère le problème du calcul des nombres $s_0, s_1, \ldots, s_{n-1}$. On suppose faites les déclarations suivantes :

```
const
   N = ... ; // Nombre de prix
type
   T_PRIX = array[0..N-1] of REAL ;     // tableau des prix
   T_SPAN = array[0..N-1] of INTEGER ; // tableau des s_i
```

Voici une fonction qui construit le tableau `S:T_SPAN` des s_j à partir d'un tableau de prix `P:T_PRIX` :

```
function calcul1(const P: T_PRIX): T_SPAN ;
var
   S: T_SPAN ;
   j,k: INTEGER ;
   ok : BOOLEAN ;
begin
   for j := low(P) to high(P) do begin
      k := low(S) ;
      ok := false ;
      while (k <= j) and not(ok) do begin
         if P[j-k] <= P[j] then k := k+1 else ok := true ;
      end; {while}
      S[j] := k ;
   end; {for}
   calcul1 := S ;
```

```
end; {function}
```

Question 2.1. *Donner l'ordre de grandeur du nombre d'évaluations des conditions du* `while` *et du* `if`.

Question 2.2. *En déduire un ordre de grandeur de la complexité de la fonction* `calcul1`.

On remarque que s_j, pour un j donné, peut être calculé facilement si l'on connait le jour le plus proche précédent j et tel que le prix pour ce jour est plus grand que le prix du jour j. Si un tel jour existe pour un jour j, on le notera $h(j)$ sinon on prendra $h(j) = -1$. Alors, dans ces conditions $s_j = j - h(j)$.

On va utiliser une pile comme structure de données auxiliaire pour stocker les jours $j, h(j), h(h(j)), \ldots$ Quand on passe du jour $j-1$ au jour j, on dépile tous les jours dont les prix sont inférieurs ou égaux à p_j, puis on empile j.

Question 2.3. *Écrire une fonction qui calcule le tableau des s_j en utilisant la remarque précédente et une pile comme structure de données auxiliaire, ce calcul étant fait en un temps linéaire en fonction de n.*

Exercice 3.

On souhaite compter le nombre d'occurrences de chacun des mots d'un texte. Par exemple, le texte

`la marinière se lasse de la vue des berges`

comprend les huit mots `la`, `marinière`, `se`, `lasse`, `de`, `vue`, `des` et `berges`, chacun d'entre eux apparaissant une fois, sauf le premier qui a deux occurrences.

Pour cela on va organiser cet ensemble de mots sous la forme d'un arbre binaire dont les nœuds sont étiquetés par les mots du texte accompagnés de leur nombre d'occurrences dans le texte, cet arbre étant ordonné selon l'ordre alphabétique des mots. Avec l'exemple de texte ci-dessus, on peut avoir l'arbre de la figure 1.

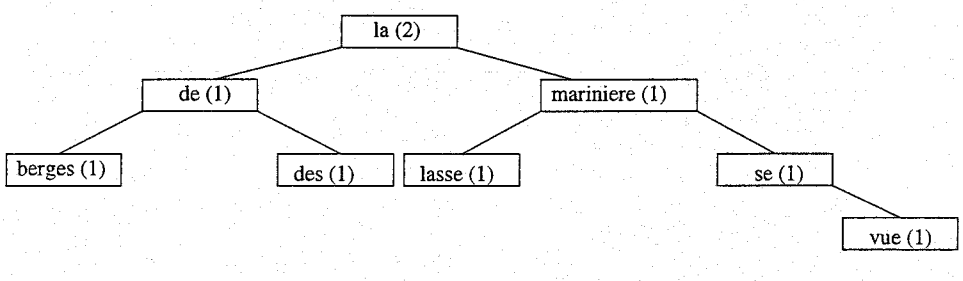

FIG. 1 – Arbre ordonné des occurences de mots

On suppose dans cet exercice que le texte qu'on examine est représenté en mémoire par une chaîne de caractères (`STRING`), ainsi que les mots.

Question 3.1. *Définir un type* `ELEMENT` *pour représenter un élément formé d'un mot et de son nombre d'occurrences.*

Dans la suite, on suppose que l'on dispose d'une procédure nommée `lireMot` dont la spécification est la suivante :

```
// lireMot(t,p,m) cherche le prochain mot m dans le texte t
// à partir de l'indice p.
// Apres l'appel à lireMot le paramètre p contient l'indice
// du caractère qui suit le mot m trouvé,
// ou bien \lstinline!length(t)+1! s'il s'agit du dernier mot.
procedure lireMot(const t: STRING; var p: CARDINAL; out m:STRING);
```

Question 3.2. *Écrire une fonction nommée* `ajouteMot`*, paramétrée par un mot et un ABO qui, selon le cas, insère un élément (de type* `ELEMENT`*) contenant ce mot, ou met à jour son compteur d'occurrences s'il est déjà présent dans l'arbre. Cette fonction renvoie un ABO.*

Question 3.3. *Écrire une fonction nommée* `occurrencesArbre` *qui construit et retourne l'ABO des mots qui composent un texte* `t`*. Chaque mot doit apparaître dans l'arbre une seule fois avec son nombre d'occurrences.*

Question 3.4. *Dessinez l'arbre construit par votre fonction appliquée au texte*

`la sœur du notaire a volé la terre du noceur`