CENTRALE-SUPELEC 97

INFORMATIQUE

PARTIE 1 - Recherche de motif

I.B.
I.B.1) Considérons l'indice k initialisé à 0 et incrémenté de une unité à chaque passage à la ligne 10 (c'est à dire retour à la première lettre du motif).
On a toujours : $k = j - i$: c'est vrai au début, puisque : $k = i = j = 0$, et si c'est vrai en entrée de boucle, alors soit j et i sont incrémentés de 1 et k ne change pas, donc on a toujours $k = (j + 1) - (i + 1)$ soit k devient $k' = k + 1$, j devient $j' = j - i + 1$ et i devient $i' = 0$, $k' = k + 1 = j - i + 1 = j' - i'$.
On peut considérer que l'on recommence alors la recherche sur le source privé de ses k premiers caractères (c'est une amorce de la méthode recursive).
En sortie de boucle, les k premiers éléments de source sont donc "rejetés", et si $i > 0$, les éléments de source compris entre k et $k + i - 1 = j - 1$ coïncident avec les éléments de motifs compris entre 0 et $i - 1$.

I.B.2) A chaque passage dans la boucle , soit i augmente (6) (mais reste inférieur à la longueur p du motif), soit k augmente (mais reste inférieur à j donc à la longueur n du source). Donc, la boucle ne peut en aucun cas être effectué plus de np fois, ce qui prouve la terminaison de l'algorithme (On verra plus loin une amélioration de cette majoration).
Lorsque l'on sort de la boucle, soit $i = p$ et les éléments compris entre k et $k + p - 1$ de source correspondent au motif : on a trouvé le motif dans source, soit $i < p$ et $j = n$, donc on a épuisé la source sans avoir trouvé le motif.

I.B.3) Fonction recherche_iterative_brute :

```
#let recherche_iter_brute motif source=
let p=vect_length motif and n=vect_length source and i=ref 0 and j=ref 0 in
while ( !i<p) && ( !j<n) do
     if motif.( !i)=source.( !j) then (i:= !i+1 ; j:= !j+1)
                                 else (j:= !j- !i+1 ; i:=0)   done ;
 !i=p ; ;

recherche_iter_brute : 'a vect -> 'a vect -> bool = <fun>

#let motif=[|1 ;2 ;1 ;2 ;3|] and source=[|1 ;2 ;2 ;1 ;2 ;1 ;2 ;3 ;1 ;2|]
     in recherche_iter_brute motif source ; ;
- : bool = true

#let motif=[|1 ;2 ;1 ;2 ;3|] and source=[|1 ;2 ;2 ;1 ;2 ;1 ;1 ;3 ;1 ;2|]
     in recherche_iter_brute motif source ; ;
- : bool = false
```

I.B.4) Dans le pire des cas, imaginons que l'on regarde à chaque fois les p éléments du motifs, donc à partir des positions 0, 1, ..., $n - p$ dans source, on aura effectué $p.(n - p + 1)$ comparaisons.

C'est le cas, par exemple, si source = $aaa \ldots aa$ et motif =$aaa \ldots aab$.
La complexité, dans le pire des cas, est $O(np)$.

I.C.
I.C.1) Note : conformément au texte du problème, la fonction est_préfixe est utilisée dans recherche_recursive. Pour compiler en Caml, il est bien sûr nécessaire de l'écrire avant.
Note : Je n'ai pas redéfini les opérations ListeVide, Construit, Tete et Queue, j'ai utilisé les fonctions Caml correspondantes.
Fonction recherche_recursive :

```
#let rec recherche_recursive motif source=
     if est_prefixe motif source then true
     else if source=[] then false
     else recherche_recursive motif (tl source) ; ;

recherche_recursive : 'a list -> 'a list -> bool = <fun>
```

I.C.2) Fonction est_prefixe :

```
#let rec est_prefixe motif source=
      if source=[] then false
      else if motif=[] then true
      else if hd motif<>hd source then false
      else est_prefixe (tl motif) (tl source) ; ;

est_prefixe : 'a list -> 'a list -> bool = <fun>

#let motif=[1 ;2 ;3] and source=[1 ;2 ;3 ;4 ;5] in est_prefixe motif source ; ;
- : bool = true

#let motif=[1 ;2 ;3] and source=[1 ;2 ;4 ;4 ;5] in est_prefixe motif source ; ;
- : bool = false
```

I.C.3) C'est le même résultat qu'à la question 1.B.4 puisque l'algorithme récursif est basé sur le même principe que l'algorithme itératif. Et le pire des cas étudié ci-dessus donne le même résultat.

I.D.
I.D.1) Fonction recherche_KMP :
On utilise le tableau obtenu par la fonction de calcul du tableau auxilliaire.
On initialise la recherche à l'indice k pour motif et i pour source, à chaque étape :

- si $0 \leq k < p$ et $i < n$, on compare motif.(k) et source.(i) :
  - s'ils sont égaux, on incremente i et k et on recommence
  - s'ils sont distincts, on ne change pas i, k devient tableau.(k)
- si $k = -1$, on incremente i et k prend la valeur 0, c'est à dire on incremente i et k.
- Si $k < p$ et $i = n$, on a épuisé la source sans trouver le motif
- Si $k = p$, on atrouvé la source dans le motif.

On en déduit la fonction Caml suivante :

```
#let recherche_KMP  motif source tableau=
let p=vect_length motif and n=vect_length source
and  k= ref 0 and i=ref 0  in
while ( !k<p) && ( !i<n) do
   if !k<0 or motif.( !k)=source.( !i) then (i:= !i+1 ;k:= !k+1)
   else k:=tableau.( !k)
done ;
 !k<p ; ;
```

```
recherche_KMP : 'a vect -> 'a vect -> int vect -> bool = <fun>

#let motif=[|1 ;2 ;1 ;2 ;3|] and source=[|1 ;2 ;1 ;2 ;1 ;2 ;3 ;1 ;2|] and tableau=[|-1 ;0 ;0 ;1 ;2|]
in recherche_KMP motif source tableau ; ;
- : bool = true
```

I.D.2) Le tableau auxiliaire pour le motif "abaababababaabaab" est $(-1, 0, 0, 1, 1, 2, 3, 2, 3, 4, 5, 6, 4)$.

I.D.3) Fonction calcule_tab_aux :
On initialise un tableau auxilliaire de longueur p à -1.
Ensuite, pour j variant de 1 à $p-1$, on recherche si le sous_mot de motif allant des indices $j-k$ à $j-1$ forment un préfixe de motif, et cela pour k variant de 1 à $j-1$ (on veut un préfixe strict.).
On utilise la fonction prefixe qui est la version vecteur de la fonction est_prefixe précédente.

```
#let calcule_tab_aux motif= let p=vect_length motif in
let t=make_vect (p) (-1)
in
for j=1 to p-1 do let l=ref 0 in
   for k=1 to j-1 do if prefixe (sub_vect motif (j-k) k) motif then l:=k done ;
t.(j) <- !l
done ;
t ; ;

calcule_tab_aux : 'a vect -> int vect = <fun>

#let motif=[|1 ;2 ;1 ;1 ;2 ;1 ;2 ;1 ;1 ;2 ;1 ;1 ;2|] in calcule_tab_aux motif ; ;
- : int vect = [|-1 ; 0 ; 0 ; 1 ; 1 ; 2 ; 3 ; 2 ; 3 ; 4 ; 5 ; 6 ; 4|]
```

PARTIE 2 - Recherche dans un dictionnaire

2.A.

```
#type info = {cle : int vect ; valeur : int}
and noeudinterne = Nil | Feuille of info | Noeud of (noeudinterne vect)
and karbre = noeudinterne ; ;

Le type info est d\'efini.
Le type noeudinterne est d\'efini.
Le type karbre est d\'efini.
```

2.A.1) Pour rechercher une clé, il faut parcourir l nœuds (où l représente la longueur commune des clés) et arriver à la feuille. Soit au total $l+1$, c'est à dire 1 plus la hauteur de l'arbre.

2.A.2) S'il y a n mots de longueur l sur un alphabet à k symboles, la hauteur de l'arbre est égale à l, chaque nœud interne est un tableau de longueur k. Il y en a au plus 1 à la racine, k au niveau 1, k^2 au niveau 2 etc...
Si $n = k^l$ (nombre maximum de clés de longueur l sur cet alphabet), il y aura $\dfrac{k^{l+1}-1}{k-1}$ tableaux de longueur k (les nœuds internes) et $n = k^l$ mots de longueur l (les feuilles).
Si $n \leq k$, au pire les premiers éléments sont 2 à 2 distincts,, il y aura $1 + nl$ nœuds et toujours n feuilles.

Si $k^p \leq n < k^{p+1}$, puisque à chaque niveau m, le nombre de nœuds est majoré par $max(k^m, n)$, au pire on remplit les $p+1$ premiers niveaux, soit $1 + k + \ldots + k^p$ et il reste ensuite $(l - p - 1)$ "niveaux" à n nœuds. Soit $\frac{k^{p+1} - 1}{k - 1} + (l - p - 1)n$ nœuds, et encore n feuilles.

2.B.

2.B.1) Type arbre comprimé :

```
#type nouveaunoeud = Nil | Feuille of int*info | Noeud of ((int*nouveaunoeud) vect)
and karbre_comprime=nouveaunoeud ; ;

Le type nouveaunoeud est d\'efini.
Le type karbre_comprime est d\'efini.
```

Recherche dans un arbre comprimé :

```
let recherche  =
let rec recherche_niveau niveau  cle = function
  | Nil -> false
  | Feuille (x,m) -> m.cle=cle
  | Noeud v -> let(decalage, branche)=v.(cle.(niveau)) in
               recherche_niveau (niveau + decalage) cle branche
in recherche_niveau 0 ; ;
```

2.B.2)

2.B.3) Le nombre de tableaux est alors inférieur à $n-1$.
En effet, tous les noeuds internes ont au moins deux fils. Montrons que dans un tel arbre, le nombre de noeuds internes est inférieur (strictement) au nombre de feuilles.
Un arbre reduit à une feuille ne possède aucun noeud interne.
Soit A un arbre ayant n feuilles ($n > 1$)et p noeuds internes, la racine a donc au moins deux fils. Soient A_1, A_2, ..., A_k ($k \geq 2$) les branches issues de la racines, n_i et p_i le nombre de feuilles et le nombre de noeuds internes de la branche A_i. Par induction structurelle, pour tout i, $p_i \leq n_i - 1$.

$$p = \sum_{i=1}^{k} p_i + 1 \qquad n = \sum_{i=1}^{k} n_i \qquad p \leq \sum_{i=1}^{k} (n_i - 1) + 1 = n - k + 1 \leq n - 1$$

La complexité en temps est au pire encore la même : si tous les nœuds internes ont deux fils , il faudra parcourir les l nœuds pour différencier les feuilles (On peut imaginer un dictionnaires dans lequel les mots vérifient cette propriété).
Mais, l'arbre n'est en général pas "plein", le gain en moyenne peut être considérable, par exemple dans l'arbre comprimé ci-dessus, le nombre moyen de nœuds parcourus est $\frac{1}{10}(2+3+3+2+1+2+2+2+2+2) = 2,1$ au lieu de 4.

2.C.
2.C.1) A chaque dictionnaire correspond un k-arbre. Le nombre de dictionnaires est $C_{k^l}^n$.
Pour que la profondeur soit strictement supérieure à d, il faut et il suffit qu'il existe deux mots du dictionnaire ayant les mêmes $d-1$ premiers symboles.
Si $n > k^{d-1}$ c'est toujours vrai donc $N_d = C_{k^l}^n$, sinon le nombre de dictionnaires pour lesquels tous les mots ont des préfixes de longueur $d-1$ distincts est $C_{k^{d-1}}^n k^{n(l-d+1)}$, donc les dictionnaires dont le k-arbre comprimé est de profondeur strictement supérieure à d est

$$N_d = C_{k^l}^n - C_{k^{d-1}}^n k^{n(l-d+1)} = C_{k^l}^n - \frac{k^{nl}}{k^{n(d-1)}} \frac{\prod_{i=0}^{n-1} \left(k^{d-1} - i\right)}{n!} = C_{k^l}^n - \frac{k^{nl}}{n!} \prod_{i=0}^{n-1} \left(1 - \frac{i}{k^{d-1}}\right)$$

Or $\frac{k^{nl}}{n!} \geq C_{k^l}^n$ donc :

$$N_d \leq C_{k^l}^n \left[1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{k^{d-1}}\right)\right]$$

Cette dernière inégalité étant vrai également si $n > k^{d-1}$ car l'un des termes du produit est alors nul et le deuxième membre vaut $C_{k^l}^n = N_d$

2.C.2)

$$\overline{d_n} = \sum_{d\geq 1} d(q_{d-1} - q_d) = \sum_{d\geq 0} q_d$$

$$q_d = \frac{N_d}{C_n^{k^l}} \leq 1 - \prod_{i=0}^{n} \left(1 - \frac{i}{k^{d-1}}\right) \leq 1 - e^{\frac{-n^2}{k^{d-1}}}$$

Ce dernier terme est ≤ 1 et aussi $\leq \frac{n^2}{k^{d-1}}$ (car, pour tout x, $e^x \geq 1 + x$). Donc :

$$q_d \leq min\left(1, \frac{n^2}{k^{d-1}}\right)$$

On en déduit donc :

$$\overline{d_n} \leq \sum_{d\geq 0} q_d \leq \sum_{d\geq 0} min\left(1, \frac{n^2}{k^{d-1}}\right)$$

Soit $\alpha = \log_k n$, si $d \geq 2\alpha + 1$ (c'est à dire $d \geq \lceil 2\alpha \rceil + 1$), $min\left(1, \frac{n^2}{k^{d-1}}\right) = \frac{n^2}{k^{d-1}}$, sinon, $min\left(1, \frac{n^2}{k^{d-1}}\right) = 1$, soit

$$\overline{d_n} \leq \lceil 2\alpha \rceil + \sum_{d\geq \lceil 2\alpha \rceil + 1} \frac{n^2}{k^{d-1}}$$

$$\sum_{d\geq \lceil 2\alpha \rceil + 1} \frac{n^2}{k^{d-1}} = \frac{n^2}{k^{\lceil 2\alpha \rceil}} \frac{k}{k-1} \leq \frac{k}{k-1} = O(1)$$

Donc,

$$\overline{d_n} \leq 2\lceil \log_k n \rceil + O(1)$$

car $\lceil 2\alpha \rceil \leq 2\lceil \alpha \rceil$.