ÉCOLE POLYTECHNIQUE - MP/PC - ÉPREUVE FACULTATIVE D'INFORMATIQUE 2007

Corrigé rédigé par Alain Schauber - alain.schauber@prepas.org

Version de MAPLE : Classic Worksheet Maple 10.
NB : dans ce problème, on manipule des tableaux unidimensionnels indexés de 0 à n - 1. Pour définir un tel tableau, on ne peut pas utiliser ici la fonction vector car pour cette fonction l'indexation doit commencer à 1. On va donc se tourner vers la fonction array . Mais il faut alors préciser l'indexation à la définition.

1 Compression par redondance

Question 1
Un algorithme de complexité linéaire en **n** :

```
> occurrences:=proc(t,n) local r,k;
    r:=array(sparse,0..255); #initialisation à la valeur 0 par
  défaut
    for k from 0 to n-1 do r[t[k]]:=r[t[k]]+1 od; #on parcourt t
  plutôt que r
    eval(r)
  end:
```

Question 2
Remarquons d'abord que le nom **min** est réservé en Maple. On va donc mettre une majuscule.
Le parcours du tableau des occurrences permet de tenir à jour la valeur d'occurrence minimale dans **t** , la minimalité dans [0..255] parmi toutes les valeurs possibles réalisant cette occurrence minimale étant assurée par l'inégalité stricte dans la condition **r[k] < r[m]**, car le minimum courant n'est modifié que s'il est d'occurrence strictement supérieure à celle de la cellule visitée.

```
> Min:=proc(t,n) local r,m,k;
    r:=occurrences(t,n);
    m:=0;
    for k from 1 to 255 do if r[k] < r[m] then m:=k fi  od;
    m
  end;
```

Question 3

On calcule d'abord la valeur du marqueur (supposée absente du texte **t**, rappelons-le). La taille **n'** du texte compressé est donc initialisée à 1 (à cause du marqueur placé en tête). Ensuite on effectue un parcours de **t** qui, à chaque cellule, examine si la valeur courante est isolée (dans ce cas on incrémente **n'**), si elle fait partie d'une répétition non terminée (on ne change pas **n'**) ou si elle est la fin d'une répétition (dans ce cas on augmente **n'** de 3). Il faut donc tenir à jour un booléen **répétition** permettant de savoir si la cellule courante fait partie d'une répétition non terminée.

```
> tailleCodage:=proc(t,n) local nprime,répétition,j;
    nprime:=1;
    répétition:=false;
    for j from 0 to n-1 do
      if j <> n-1 and t[j+1] = t[j] then
        répétition := true; #on est au sein d'une répétition non
  terminée
      else
        if répétition then nprime :=nprime + 3 else nprime :=
  nprime + 1 fi;
        répétition := false
      fi
    od;
    nprime
  end;
```

Question 4

On reprend la structure du programme de la question 3, mais cette fois en remplissant explicitement le tableau **tprime**. Il est donc nécessaire de tenir à jour une variable supplémentaire contenant l'indice de la cellule **i** du début de l'éventuelle répétition en cours de traitement. Pour éviter un doublon avec le booléen **répétition** du programme de la question 3, on peut prendre comme convention que si **i** est égal à **j**, c'est que la valeur de la cellule courante est isolée. Une variable **pos** permet de connaître l'indice de la prochaine cellule libre dans **tprime**.

```
> codage:=proc(t,n) local marqueur,tprime,i,j,pos;
    marqueur:=Min(t,n); #calcul du marqueur
    #puis déclaration du tableau destiné à accueillir la
  compression
    tprime:=array(0..tailleCodage(t,n)-1);
    tprime[0]:=marqueur;
    pos:=1; #position d'écriture courante dans le tableau tprime
    i:=0;
    for j from 0 to n-1 do
      if j = n-1 or t[j+1] <> t[j] then
        #on est en présence d'une valeur isolée ou de la fin d'une
  répétition
        if i <> j then #on achève une répétition
          tprime[pos]:=marqueur;
          tprime[pos+1]:=j-i;
          tprime[pos+2]:=t[j];
          pos:=pos+3;
        else #on a affaire à une valeur isolée
          tprime[pos]:=t[j];
          pos:=pos+1
        fi;
        i:=j+1
      fi
    od;
    eval(tprime)
  end;
```

2 Transformation de Burrows-Wheeler

Question 5

Il suffit de comparer en boucle et circulairement les couples de valeurs de **t** d'indices (i,j), (i+1,j+1),...,(i-1,j-1) jusqu'à obtention d'un couple de deux valeurs distinctes permettant de comparer les deux rotations. Dans ce cas on interrompt la boucle. Si celle-ci est menée à son terme sans interruption, c'est que les deux rotations sont égales, et on renvoie 0. Le mode «circulaire» du parcours est géré par une gestion des indices modulo **n**.

La fonction consiste en une boucle comportant au plus **n** tours, donc elle est bien linéaire par rapport à **n**.

```
> comparerRotations:=proc(t,n,i,j) local k;
    for k from 0 to n-1 do
      if t[(i+k) mod n] > t[(j+k) mod n] then RETURN(1) fi;
      if t[(i+k) mod n] < t[(j+k) mod n] then RETURN(-1) fi
    od;
    0
  end;
```

Question 6

On parcourt simultanément le tableau **transformé** et le tableau **r**. A la cellule n° **k**, le tableau **r** contient la **k**-ième rotation dans l'ordre lexicographique. Il faut donc mettre dans la cellule n° **k** de **transformé** la valeur de la dernière cellule de la rotation **rot[r[k]]**. Cette valeur se trouve dans le tableau **t** en position i telle que i-**k** = **n**-1 modulo **n**, soit i = **k**-1 modulo **n**. Au passage, on en profite pour repérer la cellule de **r** qui contient 1 : comme c'est la première rotation qui place en fin du texte la première valeur du texte initial, l'indice de la cellule de **r** qui contient 1 est celui de la clé, qu'on place alors en dernière position du tableau **transformé**.

```
> codageBW:=proc(t,n) local r,transformé,k;
    r:=triRotations(t,n);
    transformé:=array(0..n); #une cellule de plus pour la clé
    for k from 0 to n-1 do
      transformé[k] := t[(r[k]-1) mod n];
      if r[k]=1 then transformé[n] := k fi
    od;
    eval(transformé)
  end;
```

Question 7

Dans **codageBW**, la boucle est de complexité temporelle linéaire par rapport à **n**. Mais l'appel à la fonction **triRotations** réalise O(**n**ln(**n**)) comparaisons.
Comme la fonction **comparerRotations** est un O(**n**), **triRotations** est de complexité O(**n**^{2}ln(**n**)).
C'est donc l'appel à **triRotations** qui impose sa complexité à la fonction **codageBW**. La complexité de **codageBW** est dans le pire des cas de l'ordre de O(**n**^{2}ln(**n**)).

3 Transformation de Burrows-Wheeler inverse

Question 8

Rappelons la remarque du début de la partie 2 : les lettres de **t'** sont toujours des entiers dans les programmes demandés! Il suffit donc d'appliquer la fonction **occurrences** de la question 1 au sous-tableau de **t'** formé des **n** premières cellules. Etant donnée la simplicité de cette fonction, on peut tout aussi bien la réécrire.

```
> frequences:=proc(tprime,nprime) local r,k;
    r:=array(sparse,0..255); #initialisation à la valeur 0 par
  défaut
    for k from 0 to nprime-2 do r[tprime[k]]:=r[tprime[k]]+1 od;
    eval(r)
  end;
```

Question 9

Pour trier le tableau contenant le texte transformé, il suffit de calculer le tableau **freq** des fréquences de chaque valeur, puis de parcourir ce tableau dans l'ordre croissant des indices de cellules (donc des valeurs susceptibles d'être contenues dans le texte) en remplissant au fur et à mesure un tableau **triCars** d'autant d'occurrences de la valeur courante que la fréquence de cette valeur, lue dans **freq**
.

On effectue ainsi 256 tours de boucle, qui mises bout à bout, reviennent à faire un parcours du tableau **triCars**. La boucle principale de **triCarsDe** est donc de complexité linéaire par rapport à **n**. Comme il en est de même de la fonction **frequences**, on est donc assuré que la complexité de la fonction **triCarsDe** est linéaire par rapport à **n**.

```
> triCarsDe:=proc(tprime,nprime) local triCars,freq,pos,k,i;
    triCars:=array(0..nprime-2);
    freq:=frequences(tprime,nprime);
    pos:=0;
    for k from 0 to 255 do
      for i from 0 to freq[k]-1 do triCars[pos+i]:=k od;
      pos:=pos+freq[k]
    od;
    eval(triCars)
  end;
```

Question 10

Après avoir construit à partir de **t'** le tableau **triCars**, on effectue un parcours de ce dernier. Pour chaque cellule visitée de **triCars**, on parcourt le texte **t'** afin de trouver l'indice **i** de la valeur de **t'** pointée par la flèche du graphique à partir de cette cellule. Puis on renseigne la cellule n° **i** du tableau **indices** avec la valeur courante de **triCars** (égale à celle de la cellule n° **i** de **t'**). Une séquence cellules consécutives d'égale valeur de **triCars** sera gérée par une boucle imbriquée, car le nombre **m** de ces cellules consécutives peut être lu dans le tableau **freq** des fréquences de **t'**.

La fonction **trouverIndices**, après les calculs (en temps linéaire) de **triCars** et **freq**, consiste en une

boucle conditionnelle (while **i** ...) contenant une boucle imbriquée (for **j** ...) dont le résultat de chacun des **m** tours est dans le pire des cas un parcours des **nprime**-1 = **n** premières cellulesde **t'**. Or la boucle principale contient un nombre de tours tels que la somme totale des valeurs de **m** correspondantes est égale à **nprime**-1 = **n**. Il en résulte que cette boucle while est de complexité quadratique par rapport à **n** et impose donc sa complexité à l'ensemble. La complexité de la fonction **trouverIndices** est en $O(n^2)$.

```
> trouverIndices:=proc(tprime,nprime)
    local indices,freq,triCars,i,m,j,compteur,k;
    indices:=array(0..nprime-2);
    freq:=frequences(tprime,nprime);
    triCars:=triCarsDe(tprime,nprime);
    i:=0; #position de début de la séquence courante de valeurs
  égales dans triCars
    while i <= nprime-2 do
      m:=freq[triCars[i]]; #nombre d'occurrences dans t' de la
  valeur courante v de triCars
      for j from 1 to m do #on cherche dans t' l'indice k de la
  valeur de rang j égale à v
        compteur:=j; #compte à rebours du nombre de valeurs v à
  rencontrer dans t'
        k:=0;
        while compteur <> 0 do #parcours de t' jusqu'à trouver la
  cellule d'indice k
          if tprime[k] = triCars[i] then
            if compteur = 1 then indices[i+j-1]:=k fi; #k est
  l'indice cherché
            compteur:=compteur-1
          fi;
          k:=k+1
        od
      od;
      i:=i+m #la séquence courante de valeurs égales est traitée,
  on passe à la suivante
    od;
    eval(indices)
  end;
```

Question 11

Après avoir construit à partir de **t'** le tableau **indices**, on effectue un parcours de ce dernier. En partant de la cellule de **t'** pointée par la clé (qu'on peut donc placer en début du texte **décodé**, on récupère itérativement dans le tableau **indices** la position **pos** de la valeur suivante du texte **décodé** et on la récupère dans **t'** pour la placer dans **décodé**. Une boucle impérative convient puisqu'on connaît le nombre **n** = **nprime**-1 de valeurs à traiter.

La fonction **decodageBW**, après le calcul en temps quadratique de **indices**, consiste donc en cette boucle impérative de longueur **n**. La complexité de la fonction est **decodageBW** en $O(n^2)$.

```
> decodageBW:=proc(tprime,nprime) local décodé,indices,pos,k;
    décodé:=array(0..nprime-2);
    indices:=trouverIndices(tprime,nprime);
    pos:=tprime[nprime-1];
    for k from 0 to nprime-2 do
      décodé[k]:=tprime[pos];
      pos:=indices[pos]
    od;
    eval(décodé)
  end;
```

Remarque : on peut facilement montrer que la compression par redondance et la décompression inverse sont de complexité linéaire en **n**. Il en résulte que l'algorithme de compression bzip (intégrant la transformation de BW) est de complexité $O(\mathbf{n}^2\ln(\mathbf{n}))$, tandis que la décompression est de complexité $O(\mathbf{n}^2)$.
L'ensemble du processus compression/décompression est donc de complexité $O(\mathbf{n}^2\ln(\mathbf{n}))$.