Épreuve de l'option informatique

INFORMATIQUE

Concours Blanc 2024

DURÉE DE L'ÉPREUVE : 4 heures

N.B. : le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction. Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

RAPPEL DES CONSIGNES

- Utiliser uniquement un stylo noir ou bleu foncé non effaçable pour la rédaction de votre composition ; d'autres couleurs, excepté le vert, peuvent être utilisées, mais exclusivement pour les schémas et la mise en évidence des résultats.
- Ne pas utiliser de correcteur.
- Écrire le mot FIN à la fin de votre composition.

L'usage de la calculatrice est interdite.

Le sujet est composé de deux parties, toutes indépendantes.

Correction

Partie I Diamètre d'un graphe

Extrait du sujet E3A 2019

Dans cette partie, on considère des graphes non orientés connexes. Les sommets d'un graphe à n sommets ($n \in \mathbb{N}$) sont numérotés de 0 à $n-1$. On suppose qu'aucune arête ne boucle sur un même sommet.

Un chemin de longueur $p \in \mathbb{N}$ d'un sommet a vers un sommet b dans un graphe est la donnée de $p+1$ sommets $s_0, s_1, \cdots, s_p$ tels que $s_0 = a$, $s_p = b$ et, pour tout $1 \leqslant k \leqslant p$, les sommets $s_k - 1$ et s_k sont reliés par une arête.

Un plus court chemin d'un sommet a vers un sommet b dans un graphe G est un chemin de longueur minimale parmi tous les chemins de a vers b. Sa longueur est notée $d_G(a,b)$.

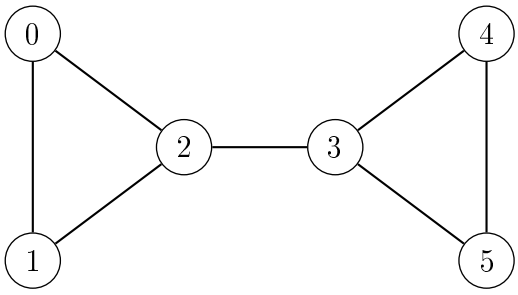
Le diamètre d'un graphe G, noté $\text{diam}(G)$, vaut le maximum des longueurs des plus courts chemins entre deux sommets du graphe G. Autrement dit,

$$\text{diam}(G) = \underset{a,b \text{ sommets de } G}{Max} d_G(a,b)$$

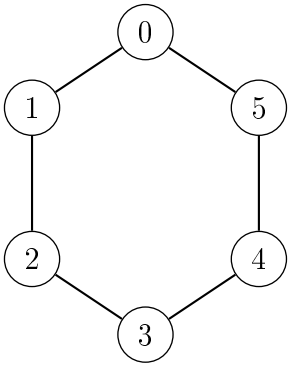
Un chemin maximal d'un graphe G est un plus court chemin de G de longueur $\text{diam}(G)$.

1 Exemples de graphes

Q-1- Donner sans justification le diamètre et les chemins maximaux pour chacun des deux graphes G_1 et G_2 ci-dessous.



graphe G_1



graphe G_2

Correction

G_1 a pour diamètre 3 et pour chemins maximaux 0, 2, 3, 4 ; 0, 2, 3, 5 ; 1, 2, 3, 4 ; 1, 2, 3, 5.
G_2 a pour diamètre 3 et pour chemins maximaux 0, 1, 2, 3 ainsi que tous les chemins « symétriques » (il y en a beaucoup...).

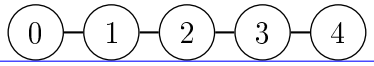
En OCaml, les graphes sont représentés par liste d'adjacence et implémentés par le type :

```
type graphe = int list array;;
```

Q-2- Graphes de diamètre maximal.

Q-2-(a) Dessiner sans justification un graphe à 5 sommets ayant un diamètre le plus grand possible.

Correction



Q-2-(b) Écrire en OCaml une fonction `diam_max` de type `int -> graphe qui prend` en argument un entier naturel n non nul et qui renvoie un graphe à n sommets de diamètre maximal.

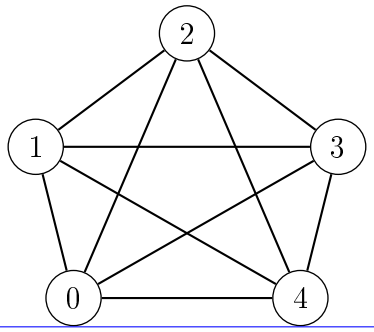
Correction

```
let diam_max n =
  let g: graphe  = Array.make n [] in
  for i = 1 to n-1 do
    g.(i-1) <- i::g.(i-1) ;
    g.(i) <- (i-1)::g.(i) ;
  done ;
  g;;
```

Q-3- Graphes de diamètre minimal.

Q-3-(a) Dessiner sans justification un graphe à 5 sommets ayant un diamètre le plus petit possible.

Correction



Q-3-(b) Écrire en OCaml une fonction `diam_min` de type `int -> graphe qui prend` en argument un entier naturel n non nul et qui renvoie un graphe à n sommets de diamètre minimal.

Correction

Il s'agit d'un graphe complet :

```
let diam_min n =
  let g: graphe  = Array.make n [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i <> j then
```

```
            g.(i) <- j::g.(i) ;
          done ;
        done ;
        g;;
```

2 Calcul du diamètre d'un graphe

On propose la fonction en Ocaml suivante :

```
let tableau_distance g depart =
  let n = Array.length g in
  let deja_visite = Array.make n false in
  deja_visite.(depart) <- true ;
  let distance = Array.make n 0 in
  let file = ref [depart] in

  while not ( !file = [] ) do
    let sommet_courant = List.hd !file in
    let rec aux liste = match liste with
        | [] -> ()
        | s::reste ->
          if deja_visite.(s) then aux reste
          else
            begin
              file := !file @ [s] ;
              distance.(s) <- distance.(sommet_courant) +1 ;
              deja_visite.(s) <- true ;
              aux reste
            end
          in
      aux g.(sommet_courant) ;
      file := List.tl !file ;
    done ;
distance ;;
```

Q-4- Quel type de parcours est définie dans la fonction `tableau_distance` ?

Correction

Il s'agit d'un parcours en largeur.

Q-5- Que renvoie la fonction `tableau_distance` avec le graphe G_1 et le départ 0 et avec le graphe G_2 et le départ 0 ?

Correction

Pour le graphe G_1, la fonction renvoie

`[0 ; 1 ; 1 ; 2 ; 3 ; 3]`

Pour le graphe G_2, la fonction renvoie

```
[0 ; 1 ; 2 ; 3 ; 2 ; 1]
```

Q-6- En utilisant la fonction précédente, déterminer la fonction `diametre` , de signature `graphe -> int` , renvoyant le diamètre d'un graphe donné en argument.

Correction

```
let diametre (g: graphe ) =
  let n = Array.length g in
  (* maxi_tab renvoie le maximum d'un tableau  *)
  let maxi_tab tab =
    let le_max = ref 0 in
    for i = 0 to n-1 do
      if tab.(i) >  !le_max  then  le_max  := tab.(i)
    done;
    !le_max
  in
  let diam = ref 0 in
  for i = 0 to n-1 do
    let max_en_i = maxi_tab (tableau_distance g i) in
    (*  On garde la plus grande des distances en partant de i *)
    if max_en_i > !diam then
      diam := max_en_i
  done ;
  !diam ;;
```

Q-7- Déterminer la complexité dans le pire des cas de la fonction `diametre` en fonction du nombre de sommets.

Correction

Dans le pire des cas, un graphe contient un voisinage de n^2 arêtes, pour n sommets. La complexité de la fonction `tableau_distance` est donc en $O(n^2)$. (en fait pour chaque sommets, on parcours l'ensemble de ses arêtes.)
On obtient donc une complexité en $O(n^3)$ pour la fonction `diametre`.

3 Diamètre d'un arbre binaire

Dans cette section, on s'intéresse aux arbres binaires, qui sont des cas particuliers de graphes. On travaille avec une représentation spécifique de ces graphes particuliers, implémentée en OCaml par le type suivant :

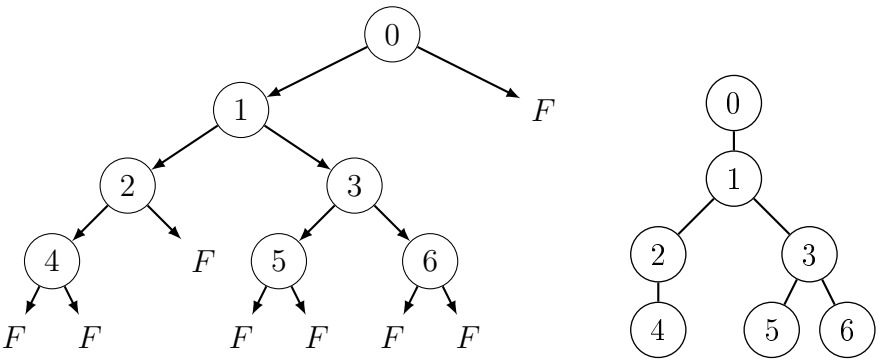
```
type arbre = Feuille | Noeud of int * arbre * arbre ;;
```

Le graphe sous-jacent G_A à un arbre binaire A est défini comme le graphe orienté dont

- les sommets correspondent aux nœuds de l'arbre (et pas aux feuilles) ;
- les arêtes correspondent aux branches de l'arbre reliant deux nœuds (et non pas celles reliant un nœud à une feuille).

Le diamètre d'un arbre binaire est alors défini comme le diamètre de son graphe sous-jacent.

Voici un exemple d'arbre binaire A (à gauche) et de son graphe sous-jacent G_A (à droite) :



Q-8- Donner l'expression OCaml représentant l'arbre A de l'exemple. Donner le diamètre de A et les chemins maximaux du graphe sous-jacent G_A.

Correction

```
let exemple_A =
Noeud(0 , Noeud(1 , Noeud(2,Noeud(4,Feuille,Feuille) , Feuille) , Noeud
(3, Noeud(5 , Feuille,
Feuille) , Noeud(6 , Feuille , Feuille))), Feuille);;
```

Le diamètre est de 4 ; les chemins maximaux sont 4 ;2 ;1 ;3 ;6 ou 4 ;2 ;1 ;3 ;5.

Q-9- Quel est le nombre r d'arêtes du graphe sous-jacent à un arbre binaire possédant n nœuds ($n > 0$) ?

Correction

Notons $r(A)$ le nombres d'arêtes du graphe sous-jacent à l'arbre binaire A, et $n(A)$ son nombre de nœud. Si A est une feuille, on impose alors $r(A) = -1$.
Nous allons montrer, par induction structurelle, la proposition suivante, pour A un arbre binaire :

$$\mathcal{P}(A) : "r(A) = n(A) - 1"$$

- Si A n'a qu'un seul nœud, G_A est réduit à ce seul sommet, donc $r(A) = 0 = n(A) - 1$. La proposition est donc vraie.
- On suppose que $A = Noeud(racine, Ag, Ad)$ tel que la proposition est vraie pour Ag et Ad.
  Par hypothèse, $r(Ag) = n(Ag) - 1$ et $r(Ad) = n(Ad) - 1$. De plus, par construction du graphe sous-jacent, $r(A) = r(Ag) + r(Ad) + 2$ (les deux arêtes reliant la racine aux graphe sous-jacent des arbre Ag et Ad. La formule reste vraie si l'un des arbres est une feuille). De plus, on a $n(A) = n(Ad) + n(Ag) + 1$.

  r(A) = r(Ag) + r(Ad) +2
  = n(Ad) -1 + n(Ad) -1 + 2
  = n(Ad) + n(Ad)
  = n(A) -1

  On a donc la propriété pour A.

Ce qui termine la démonstration par induction.
On peut donc répondre à la question par :

$$r = n - 1$$

Une première approche pour calculer le diamètre d'un arbre consiste à le transformer en un graphe et à employer un algorithme général sur les graphes de la partie 2.

Q-10- Écrire en OCaml une fonction `nb_noeuds` de type `arbre -> int` qui renvoie le nombre de nœuds d'un arbre binaire donné en argument.

Correction

```
let rec nb_noeuds arbre = match arbre with
  | Feuille -> 0
  | Noeud(racine, ag , ad ) -> 1 + nb_noeuds ag + nb_noeuds ad ;;
```

Q-11- Écrire en OCaml une fonction `numerotation` de type `arbre -> arbre` qui prend en argument un arbre binaire A à n nœuds et qui renvoie un arbre binaire A' de même graphe sous-jacent que A et dont les nœuds sont étiquetés de 0 à $n-1$.

Correction

On peut utiliser la fonction précédente :

```
let numerotation arbre =
  let rec aux i a = match a with
    | Feuille -> Feuille
    | Noeud(racine, ag , ad) -> let nb_ag = nb_noeuds ag in
      let new_ag = aux (i+1) ag in
      let new_ad = aux (i +1 +nb_ag ) ad in
        Noeud( i ,new_ag , new_ad )
  in aux 0 arbre ;;
```

Mais la fonction parcours plusieurs fois l'arbre. Une autre méthode, moins couteuse, permet de récupérer la taille de l'arbre, tout en construisant l'arbre recherché :

```
let  numerotation_2 arbre =
  let rec aux i a = match a with
  |Feuille -> (Feuille , 0)
  | Noeud(racine , ag , ad ) ->
    let ( new_ag , nb_g ) = aux (i+1)  ag in
    let ( new_ad , nb_d ) = aux  (i+nb_g +1) ad in
    (Noeud( i , new_ag , new_ad ) , 1+ nb_g + nb_d )
  in
  let (result, total ) = aux 0 arbre   in
  result ;;
```

Q-12- Écrire en OCaml une fonction `arbre_vers_graphe` de type `arbre -> graphe` qui prend en argument un arbre binaire A à n nœuds étiquetés de 0 à $n-1$ et qui renvoie le graphe G_A sous-jacent à A.

Correction

```
let arbre_vers_graphe arbre =
  let n =  nb_noeuds arbre in
  let (g : graphe ) = Array.make n [] in
  let rec aux a = match a with
    | Feuille -> ()
    | Noeud( i , Feuille , Feuille ) -> ()
    | Noeud( i , a_g , Feuille ) -> let j = racine a_g in
      g.(i) <- j::g.(i) ;  g.(j) <- i::g.(j) ; aux a_g
    | Noeud( i , Feuille , a_d ) -> let j = racine a_d in
      g.(i) <- j::g.(i) ;  g.(j) <- i::g.(j) ; aux a_d
    | Noeud( i , a_g , a_d ) -> let j = racine a_d in
    let k = racine a_g in
      g.(i) <- j::g.(i) ;  g.(j) <- i::g.(j) ;
      g.(i) <- k::g.(i) ;  g.(k) <- i::g.(k)  ; aux a_d ; aux a_g
    in
      aux arbre ;
    g;;
```

Q-13- Écrire la fonction un algorithme `diametre_arbre` qui calcule le diamètre d'un arbre de type arbre en se ramenant à un graphe.
Quelle est sa complexité ?

Correction

On rassemble toute les fonctions précédentes.

```
let diametre_arbre arbre =
  let a_prime = arbre_vers_graphe (numerotation arbre) in
    diametre a_prime ;;
```

La complexité n'est en réalité donnée que par la fonction diamètre. (les autres étant négligeables devant cette dernière...). On trouve donc une complexité en $O(n^3)$.

Une seconde approche pour calculer le diamètre d'un arbre consiste à employer une technique « diviser-pour-régner » .
Pour tout arbre A non réduit à une feuille, de la forme `Noeud (x, arbre_g, arbre_d)`, on note

- A_g le fils gauche de A, représenté par `arbre_g` ;
- A_d le fils droit de A, représenté par `arbre_d` ;

La hauteur de l'arbre A, notée $h(A)$, est la longueur du plus long chemin descendant de la racine vers une feuille. Dans l'arbre exemple de la partie 3, l'arbre A est de hauteur 4.

Q-14- Quelle est la longueur d'un chemin maximal passant par la racine ?

Correction

On va démontrer que pour A non réduit à une feuille, de la forme `Noeud (x, arbre_g, arbre_d)`, la longueur (noté $C(A)$)d'un chemin maximal passant par la racine est :

$$C(A) = h(A_g) + h(A_d)$$

(en posant, si l'arbre est une feuille, $h(A) = -1$)

En effet, si on note $C_r(A)$ le chemin de taille maximal dont une extrémité est la racine, on a $C_r(A) = h(A) - 1$, par définition de la hauteur.(cohérent aussi si l'arbre est une feuille, en posant $C_r(A) = -2$)

On a donc :
$$\begin{aligned} C(A) &= C_r(A_g) + C_r(A_d) + 2 \\ &= h(A_g) - 1 + h(A_d) - 1 + 2 \\ &= h(A_g) + h(A_d) \end{aligned}$$

Q-15- Écrire en OCaml une fonction `diam_arbre` de type `arbre -> int` qui calcule le diamètre d'un arbre donné en argument. Cette fonction devra être de complexité linéaire en le nombre de nœuds de l'arbre.

Correction

```
let diam_arbre arbre =
 let rec aux arbre = match arbre with
   | Feuille -> (-1 , 0 )
   | Noeud( r , ag , ad ) ->
     let (dia_g , haut_g ) = aux ag in
     let (dia_d , haut_d ) = aux ad in
     (max ( max dia_g dia_d ) (haut_g + haut_d) , 1 + max haut_g haut_d
)
   in
   fst(aux arbre) ;;
```

La complexité est bien linéaire...

Partie II Algorithmique des mots sans facteur carré

Extrait du sujet CCINP 2019

L'objectif de cette partie est de construire différents algorithmes pour vérifier si un mot comporte des facteurs carrés ou non.
Dans toute la suite, $\Sigma = \{a_1 < \ldots < a_p\}$ désigne un alphabet totalement ordonné comportant p lettres, ϵ représente le mot vide et Σ^* est l'ensemble des mots finis obtenus à partir de Σ. Pour tout réel x, on note $\lfloor x \rfloor$ la partie entière de x.

4 Définitions

Définition 1 (Longueur d'un mot).
Soit $w = w_0 ... w_{n-1}$ un mot de Σ^*. La longueur n de w est notée $|w|$, pour tout $0 \leqslant i \leqslant j < n$, $w[i, j]$ désigne le mot $w_i ... w_j$. Par convention, si $j < i$, $w[i, j]$ désigne ϵ.

Définition 2 (Mot carré).
Soit w un mot de Σ^*. On dit que w est un carré s'il existe un mot x tel que $w = x.x$.

Définition 3 (Facteur d'un mot).
Soient v et w deux mots de Σ^*. On dit que v est un facteur de w s'il existe r et s deux mots (éventuellement vides) tels que $w = r.v.s$.

Définition 4 (Répétition).
On dit qu'un mot w contient une répétition s'il contient un facteur carré différent de ϵ.

Dans la suite, un mot sera représenté en Caml par la liste de ses lettres. Par exemple, le mot *baba* est représenté par la liste `['b';'a';'b';'a']` et le mot vide est représenté par la liste `[]`.

5 Fonctions utiles sur les listes

Q-16- Écrire une fonction récursive Caml de signature `longueur :'a list -> int` qui renvoie la longueur de la liste.

Correction

```
let rec longueur liste = match liste with
  | [] -> 0
  | a::reste -> 1 + longueur reste ;;
```

Q-17- Écrire une fonction Caml de signature `sous_liste : 'a list -> int -> int -> 'a list` où `sous_liste L k long` renvoie une liste `S` qui est la sous-liste de `L` commençant à l'indice `k` et de longueur `long`. On suppose que l'indexation des listes commence à 0.

Correction

```
let rec sous_liste liste k long =
  match liste with
    | [] when (k>0) || long > 0 -> failwith "erreur"
    | a::reste when k> 0 -> sous_liste reste (k-1) long
    | a::reste when long > 0 -> a::(sous_liste reste 0 (long-1))
    | _ -> [];;
```

On pourra dans la suite de l'énoncé utiliser les fonctions `longueur` et `sous_liste`.

6 Un algorithme naïf

Q-18- Préciser si les mots suivants contiennent ou non une répétition.

(i) **aabfa** | (ii) **abfdanq** | (iii) **ababa** | (iv) **avba**

Correction

(i) et (iii) contiennent des facteurs carrés, mais pas (ii) et (iv)...

Q-19- Soit w un mot contenant au plus deux lettres différentes. Montrer que si $|w| \geqslant 4$ alors w contient au moins une répétition.

Correction

Si w contient une seule lettre et $|w| \geqslant 2$, il a contient forcément une facteur carré.
Supposons que w contienne deux lettres distinctes a et b, si deux lettres se suivent, il y a donc un facteur carré, sinon, si les lettres n'alternent (comment $abab$) il y a donc aussi un facteur carré.

Q-20- Écrire une fonction Caml de signature `estCarre : 'a list -> bool` prenant en argument une liste w et retournant `true` si w est un carré et `false` sinon.

Correction

Si le mot n'a pas une longueur paire, il ne peut être un carré. Sinon, on récupère la première moitié et la dernière moitié, et on les compare.

```
let estCarre mot =
  let n = longueur mot in
  if n mod 2 = 1 then false
  else
    begin
      let moitier_inf =  sous_liste mot 0 (n/2) in
      let moitier_sup =  sous_liste mot (n/2) (n/2) in
      moitier_inf = moitier_sup
    end;;
```

Q-21- Déterminer la complexité de la fonction `estCarre`.

Correction

La complexité de la fonction `longueur` est en $O(n)$, ainsi que les fonctions `sous_liste`...
On reste donc sur une complexité en $O(n)$.

Q-22- Écrire une fonction Caml de signature `contientRepetitionAux : 'a list -> int -> bool` prenant en argument une liste w et un entier m et retournant `true` si w contient une répétition de la forme `xx` avec `x` de longueur m et `false` sinon.

Correction

L'idée est de parcourir tous les sous-mots de longueur $2m$ et de vérifier s'il sont des carrés.

```
let contientRepetitionAux mot m =
  let n = longueur mot in
  let booleen = ref false in
  for i = 0 to n-2*m do
    let sous_mot = sous_liste mot i (2*m) in
    if estCarre sous_mot then booleen := true
  done;
  !booleen ;;
```

On obtient en complexité en $O(n^2)$.

Q-23- Montrer que toute répétition d'un mot w de longueur n est de la forme xx avec $|x| \leqslant \frac{n}{2}$.

Correction

On a évidement $|xx| \leqslant n \Leftrightarrow 2|x| \leqslant n \Leftrightarrow |x| \leqslant \frac{n}{2}$.

Q-24- En déduire une fonction Caml de signature `contientRepetition : 'a list -> bool` prenant en argument une liste w retournant `true` si w contient une répétition et `false` sinon.

Correction

L'idée est de chercher les sous-mot carré de taille de 1 à $n/2$...

```
let contientRepetition mot =
  let n = longueur mot in
  let booleen = ref false in
  for m = 1 to n/2 do
    if contientRepetitionAux mot m then booleen := true
  done ;
  !booleen ;;
```

Q-25- Quelle est la complexité de la fonction `contientRepetition` ?

Correction

On a donc une complexité en $O(n^3)$...

7 Algorithme de Main-Lorentz

L'algorithme de Main-Lorentz permet de détecter de manière plus efficace des répétitions d'un mot w. Il comporte essentiellement deux parties :

- la première consiste à voir si étant donné deux mots u et v, le mot uv contient un carré non nul issu de la concaténation ;
- la deuxième s'appuie sur le principe de "diviser pour régner".

Remarquons qu'un mot uv contient une répétition si et seulement si u ou v contiennent une répétition ou uv contient des répétitions provenant de la concaténation. Pour déterminer si un mot uv contient de nouvelles répétitions, on commence par effectuer des pré-traitements consistant à calculer des tables de valeurs de u et de v qui sont généralement appelées tables de préfixes (ou suffixes). Avant de présenter des algorithmes permettant de générer ces tables, on commence par justifier leur application dans la détection de répétitions.

Définition 13 (Carré centré).
Soient u et v deux mots. On dit que uv contient un carré centré sur u (respectivement

sur v) s'il existe un mot w non vide et des mots u' , v'', w' , w'' tels que $u = u'ww'$, $v = w''v''$, $w = w'w''$(respectivement $u = u'w'$, $v = w''wv''$, $w = w'w''$).

Définition 14 (Plus long préfixe commun, plus long suffixe commun).
Soient u et v deux mots. Le plus long préfixe (respectivement suffixe) commun de u et v est le plus long mot w tel qu'il existe deux mots r et s tels que $u = wr$ et $v = ws$ (respectivement $u = rw$ et $v = sw$). On le note $lcp(u, v)$ (respectivement $lcs(u, v)$).

À propos des carrés centrés :

Q-26- Dans cette question, $\Sigma = \{a, b\}$. Soient $u = abababaa$ et $v = ababaaa$. Déterminer le plus long préfixe commun de u et v.

Correction

$lcp(u, v) = ababa...$

Q-27- Soient u et v deux mots de Σ^*. Montrer que uv contient un carré centré sur u si et seulement s'il existe $i \in \{0, ..., |u| - 1\}$ tel que

$$|lcs\,(u[0, i-1], u)\,| + |lcp\,(u[i, |u|-1], v)\,| \geqslant |u| - i$$

Correction

- **CN** : Supposons que uv contient un carré centré sur u :

$$\underbrace{u'(w'w'')(w'}_{u}\underbrace{w'')v'}_{v}$$

En posant $i = |u'w'|$, on a w' est un suffixe commun à $u[0, i-1]$ et u, donc $|lcs\,(u[0, i-1], u)\,| \geqslant |w'|$, et w'' est un préfixe commun à $u[i, |u|-1]$ et à v, donc $|lcp\,(u[i, |u|-1], v)\,| \geqslant |w''|$. Ce qui donne :

$$\begin{array}{rcl} |lcs\,(u[0, i-1], u)\,| + |lcp\,(u[i, |u|-1], v)\,| & \geqslant & |w'| + |w''| \\ & \geqslant & |u'w'| + |w'| + |w''| - i \\ & \geqslant & |u| - i \end{array}$$

- **CS** : Supposons l'existence du i. On note alors $w' = lcs\,(u[0, i-1], u)$. On a donc l'inégalité :
$|lcp\,(u[i, |u|-1], v)\,| \geqslant |u| - i - |w'|$.
On peut donc trouver un préfixe w'' commun à $u[i, |u|-1]$ et v (pas forcément le plus grand) mais vérifiant :
$|w''| = |u| - i - |w'|$ soit $|w''| + |w'| = |u| - i = |u[i, |u|-1]|$.
Avec w'' un préfixe de $u[i, |u|-1]$, et w' un suffixe. on obtient l'existence de u' tel que $u'w' = u[0, i-1]$ et $w''w' = u[i, |u|-1]$, soit $u = u'w'w''w'$.
On trouve enfin la décomposition de v.

De la même manière, on peut montrer que uv contient un carré centré sur v si et seulement s'il existe $j \in \{0, ..., |v| - 1\}$ tel que

$$|lcs\,(v[0, j-1], u)\,| + |lcp\,(v, v[j, |v|-1])\,| \geqslant |v| - j$$

Ainsi, pour pouvoir déterminer s'il existe un carré centré sur u ou v, on peut utiliser les valeurs :

$$|lcs\,(u[0,i-1],u)\,|,|lcp\,(u[i,|u|-1],v)\,|,|lcs\,(v[0,j-1],u)\,|,|lcp\,(v,v[j,|v|-1])\,|$$

Dans la suite, étant donné deux mots u et v, on note pref_u, $\text{pref}_{u,v}$, suff_u et $\text{suff}_{u,v}$ les tableaux vérifiant :

$$\forall i\in\{0,...,|u|-1\}\quad \text{pref}_u[i]=|lcp\,(u[i,|u|-1],u)\,|\quad,\quad \text{pref}_{u,v}[i]=|lcp\,(u[i,|u|-1],v)\,|$$

$$\text{suff}_u[i]=|lcs\,(u[0,i],u)\,|\quad,\quad \text{suff}_{u,v}[i]=|lcs\,(u[0,i],v)\,|$$

Calcul de table de préfixes

On présente un algorithme permettant le calcul de la table pref_u ainsi que sa complexité en nombre de comparaisons de caractères. En adaptant cet algorithme, il est également possible de calculer la table $\text{pref}_{u,v}$ en $O(|u|)$ de comparaisons de caractères.

Q-28- On pose $u=aabbba$ et $v=abbaab$. Déterminer les tableaux pref_u et $\text{pref}_{u,v}$ sans justification.

Correction

$\text{pref}_u=[|6;1;0;0;0;1|]$ et $\text{pref}_{u,v}=[|1;3;0;0;0;1|]$

Algorithme : Calcul de la table pref_u

Entrées : une chaîne de caractères u.
Sorties : un tableau pref_u

$i \leftarrow 0$, pref $\leftarrow$ tableau de taille $|u|$ initialisé à 0, pref$[i] \leftarrow |u|$, $g \leftarrow 0$
pour *i allant de 1 à* $|u|-1$ **faire**
  si $i < g$ *et* pref$[i-f] < g-i$ **alors**
    pref$[i] \leftarrow$ pref$[i-f]$
  fin
  sinon si $i < g$ *et* pref$[i-f] > g-i$ **alors**
    pref$[i] \leftarrow g-i$
  fin
  sinon
    $(f,g) \leftarrow (i, \max(g,i))$
    tant que $g < |u|$ *et* $u[g] == u[g-f]$ **faire**
      $g \leftarrow g+1$
    fin
    pref$[i] \leftarrow g-f$
  fin
fin

On admet que la complexité est de $O(|u|)$ en nombre de comparaisons de caractères.

Q-29- En déroulant l'algorithme précédent appliqué au mot $u = aaabaaabaaab$, compléter le tableau de la façon suivante : pour une valeur i donnée, on indique les valeurs de f, g, pref$[i]$ à l'issue des instructions internes de la boucle.

Par exemple, à l'initialisation, $i = 0$, f n'est pas définie, g vaut 0 et pref$[0] = 12$. Pour $i = 1$, à l'issue des instructions internes à la boucle, on a $f = 1$, $g = 3$, pref$[1] = 2$.

i	f	g	pref$[i]$
0	_	0	12
1	1	3	2
2	...	...	...
⋮	⋮	⋮	⋮

Correction

i	f	g	pref$[i]$
0	_	0	12
1	1	3	2
2	2	3	1
3	3	3	0
4	4	12	8
5	4	12	2
6	4	12	1
7	4	12	0
8	4	12	4
9	4	12	2
10	4	12	1
11	4	12	0

Q-30- Déduire de l'algorithme précédent une procédure calculant suff_u.

Correction

La construction du tableau des suffixes est obtenu simplement en utilisant le mot miroir, et en cherchant encore le tableau des préfixes.

Dans la suite, on suppose que l'algorithme `tabpref(u,v)` qui prend en argument deux chaînes de caractères u et v et qui renvoie la table $\text{pref}_{u,v}$ nous est donné. On admet que la complexité de cet algorithme est de $O(|u|)$ en nombre de comparaisons de caractères.

Q-31- Déduire des questions précédentes un algorithme qui, étant donnés deux mots u et v, renvoie `VRAI` s'il existe un carré centré sur u et `FAUX` sinon.

Correction

Q-32- Quelle est la complexité de cet algorithme en nombre de comparaisons de caractères ?

Correction

Le calcul des deux tableaux $\text{pref}_{u,v}$ et suff_u se fait en $O(n)$. La boucle coûte aussi $O(n)$. La complexité est donc en $O(n)$.

Application des tables

Q-33- Déduire des questions précédentes un algorithme récursif qui prend en argument une chaîne de caractères et renvoie `VRAI` si la chaîne contient une répétition et `FAUX` sinon.

Correction

Q-34- Déterminer la complexité de cet algorithme en nombre de comparaisons de caractères.

Correction

FIN