Corrigé du DS3

Identification de facteurs répétés dans un mot

Partie A – Préliminaires sur les vecteurs et piles

A.1 Pensez à bien initialiser votre vecteur afin de respecter le typage demandé.

```
# let map_vect f t = let n = vect_length t in
        let result = make_vect n (f t.(0)) in
                for i = 1 to n - 1 do
                        result.(i) <- f t.(i)
                done;
        result;;
map_vect : ('a -> 'b) -> 'a vect -> 'b vect = <fun>
```

A.2 Même remarque qu'à la question précédente.

```
# let iter_vect f debut fin = let n = fin - debut + 1 in
        let result = make_vect n (f debut) in
                for i = debut + 1 to fin do
                        result.(i) <- f i
                done;
        result;;
iter_vect : (int -> 'a) -> int -> int -> 'a vect = <fun>
```

Remarque : on aurait pu utiliser map_vect pour définir iter_vect , et réciproquement.

A.3

```
# let conversion s = let l = string_length s in
        iter_vect (function i -> numero_lettre s.[i]) 0 (l - 1);;
conversion : string -> int vect = <fun>
```

A.4

```
# let cree_vide () = {elements = []};;
cree_vide : unit -> pile = <fun>

# let est_vide p = p.elements = [];;
est_vide : pile -> bool = <fun>

# let depile p = match p.elements with
        | [] -> failwith "on ne depile pas la pile vide"
        | a :: q -> p.elements <- q; a;;
depile : pile -> int = <fun>

# let empile a p = p.elements <- a :: p.elements;;
empile : int -> pile -> unit = <fun>
```

Partie B – Première approche

B.1

```
# let facteurs s a = map_vect conversion
(iter_vect (function i -> sub_string s i a) 0 (string_length s - a));;
facteurs : string -> int -> int vect vect = <fun>
```

B.2

a

```
# let classesEq s a = let temp = facteurs s a in
        let l = vect_length temp in
        let res = make_vect l 0 in
                for i = 1 to l - 1 do
                let vect = temp.(i) and j = ref 0 and indice_max = ref 0 in
                                while !j < i && temp.(!j) <> vect do
                                        indice_max := max !indice_max res.(!j);
                                        j := !j + 1
                                done;
                res.(i) <- if !j = i then !indice_max + 1 else res.(!j)
                done;
        res;;
classesEq : string -> int -> int vect = <fun>
```

b La complexité est en n^2a : dans la fonction classesEq, la boucle while a un coût en ai au pire (comparaison temp.(!j) <> vect pour chaque j!) donc la boucle for a un coût en al^2 au pire, où $l = n - a$. Le reste de la fonction (notamment l'appel de facteurs) est moins coûteux. Le coût est donc en $a(n-a)^2$ au pire, donc dominé par an^2.

B.3

a $< i, E_1, j >$ signifie simplement que $x_i = x_j$.

b L'idée consiste à indexer les lettres utilisées dans le mot par leur ordre d'apparition.

```
# let construit_E1 s =
        let l = string_length s and temp_vect = make_vect 26 (-1) in
                let res = make_vect l 0 in
                        let j = ref 0 in
                                for i = 0 to l - 1 do
                                        let k = numero_lettre s.[i] in
                                                if temp_vect.(k) = - 1 then
                                                        (temp_vect.(k) <- !j;
                                                        j := !j + 1)
                                done;
                for i = 0 to l - 1 do
                        res.(i) <- temp_vect.(numero_lettre s.[i])
                done;
        res;;
construit_E1 : string -> int vect = <fun>
```

Partie C – Une stratégie diviser pour régner

C.1

a $< i, E_{a+b}, j >$ équivaut à : $< i, E_a, j > \wedge < i + a, E_b, j + a >$

b $< i, E_{a+b}, j >$ équivaut à : $< i, E_a, j > \wedge < i + b, E_a, j + b >$

C.2

a On initialise un vecteur de piles vides de taille $n - (a + b) + 1$. Pour i allant de 0 à $n - (a + b)$, on empile i sur la pile d'indice $v.(i + b)$.

b Voici une version où on n'utilise pas cree_vect_pile_vide :

```
# let empileClassesEa v = let l = vect_length v in
        let res = map_vect (function i -> {elements = [i]}) v in
                for i = 0 to l - 1 do
                        let elt = depile res.(i) in empile i res.(v.(i))
                done;
        res;;
empileClassesEa : int vect -> pile vect = <fun>
```

c

```
# let sousensembles v pile b =
        let l = vect_length v and l' = vect_length pile in
        let res = cree_vect_pile_vide l in
                for i = 0 to l' - 1 do
                        while not (est_vide pile.(i)) do
                        let k = depile pile.(i) in
                                if k + b < l then empile k res.(v.(k + b))
                        done;
                done;
        res;;
sousensembles : int vect -> pile vect -> int -> pile vect = <fun>
```

d On dépile chaque pile dans l'ordre naturel, en regardant à quel moment on change de classe d'équivalence pour E_a, grâce à notre façon de les avoir empilées.

```
# let classeAplusB v pile_vect b =
        let l = vect_length v and k = vect_length pile_vect in
        let res = make_vect (l - b) 0 in
                for i = 0 to (k - 1) do
                        if not (est_vide pile_vect.(i)) then
                        let i0 = ref (depile pile_vect.(i)) in
                                res.(!i0) <- !i0; (* mauvaise numérotation *)
                                while not (est_vide pile_vect.(i)) do
                                let k' = depile pile_vect.(i) in
                                        if v.(k') <> v.(!i0) then i0 := k';
                                        res.(k') <- !i0;
                                done;
                done;
        res;;
classeAplusB : 'a vect -> pile vect -> int -> int vect = <fun>
```

On peut donner le bon résultat (avec la bonne indexation) en utilisant un vecteur de piles supplémentaire :

```
let classeAplusB v pile_vect b =
        let l = vect_length v and k = vect_length pile_vect in
                let res = cree_vect_pile_vide l in
                        for i = 0 to (k - 1) do if not (est_vide pile_vect.(i)) then
                                let i0 = ref (depile pile_vect.(i)) in
                                        empile !i0 res.(!i0);
                                        while not (est_vide pile_vect.(i)) do
                                                let k' = depile pile_vect.(i) in
                                                if v.(k') <> v.(!i0) then i0 := k';
                                                empile k' res.(!i0);
                                        done;
                        done;
        let vrai_res = make_vect (l - b) 0 and classe = ref 0 in
                for i = 0 to l - 1 do if not (est_vide res.(i)) then
                begin
                        while not (est_vide res.(i)) do
                        let j = depile res.(i) in vrai_res.(j) <- !classe;
                        done;
                        classe := !classe + 1;
                end
                done;
        vrai_res;;
classeAplusB : 'a vect -> pile vect -> int -> int vect = <fun>
```

e Question facile, que tout le monde pouvait traiter en admettant les résultats précédents.

```
# let classeAversAplusB v b =
        classeAplusB v (sousensembles v (empileClassesEa v) b) b;;
```

```
classeAversAplusB : int vect -> int -> int vect = <fun>
```

C.3 On peut s'inspirer d'une des expressions de l'exponentiation rapide :

```
# let rec classesEq m = let l = string_length m in function
        | 1 -> construit_E1 m
        | k when k mod 2 = 0 -> classeAversAplusB (classesEq m (k / 2)) (k / 2)
        | k -> classeAversAplusB (classesEq m (k - 1)) 1;;
classesEq : string -> int -> int vect = <fun>
```