Centrale 2021

I Quelques Fonctions Auxiliaires

Q1.
```
let nombre_aretes g =
    let rec length_list l =
        match l with
        |[] -> 0
        |h::t -> 1 + length_list t
    in
    let s = ref 0 in
    for i = 0 to Array.length(g) - 1 do
        s := !s + length_list g.(i)
    done;
    !s / 2 ;;
```

Q2.
```
let g_2_3 = [|
    [|3; 1|];
    [|4; 0; 2|];
    [|5; 1|];
    [|0; 4|];
    [|1; 3 ; 5|];
    [|2; 4|];
    |];;
```

Q3.
```
let adjacence g =
    let n = Array.length g in
    let adj = Array.make n [||] in
    for i = 0 to n - 1 do
        adj.(i) <- Array.of_list g.(i)
    done;
    adj ;;
```

Q4.
```
let rang (p, q) (s, t) =
    let is, js = s / p, s mod p in
    let it, jt = t / p, t mod p in
    if it = is + 1 then
        (q - 1) * js + is
    else if jt = js + 1 then
        p * (q - 1) + (p - 1) * is + js
    else
        failwith "Argument(s) invalide(s)" ;;
```

Q5.
```
let sommets (p, q) rg =
    if rg < p * (q - 1) then
        let is, js = rg mod (q - 1), rg / (q - 1) in
        let s = is * p + js in
        (s, s + p)
    else if rg < p * (q - 1) + q * (p - 1) then
        let shift = p * (q - 1) in
        let is, js = (rg - shift) mod (p - 1), (rg - shift) / (p - 1) in
        let s = js * (q + 1) + is in
        (s, s + 1)
    else failwith "Argument(s) invalide(s)" ;;
```

Q6.
```
let quadrillage p q =
    let graphe = Array.make (p * q) [] in
    let rec remplissage_graphe rg =
        if rg < p * (q - 1) + q * (p - 1) then
            let v1, v2 = sommets (p, q) rg in
            begin
```

```
            graphe.(v1) <- v2 :: graphe.(v1) ;
            graphe.(v2) <- v1 :: graphe.(v2);
            remplissage_graphe (rg + 1);
            end
    in
    remplissage_graphe 0 ;
    graphe ;;
```

II Caractérisation des arbres

II.A - Propriétés sur les arbres

Q7. Si $s, t \in S_n$, notons $s * t$, la relation "Il existe un chemin de s à t".
Montrons que $*$ est une relation d'équivalence sur S_n.

- Réflexivité : soit $s \in S_n$. Par convention, il existe un chemin de s à s. Donc $s * s$.
- Symétrie : soit $s, t \in S_n$, si $s * t$, alors il existe un chemin $c = (s, s1, \dots, s_{k-1}, t)$. Donc $\forall i \in \{0, \dots, k-1\}$, $\{s_i, s_{i+1}\} \in A$ donc $\{s_{i+1}, s_i\} \in A$, donc le chemin $c' = (t, s_{k-1}, \dots, s_1, s)$ existe et donc $t * s$
- Transitivité : soit $s, t, u \in S_n$ tels que si $s * t$ et $t * u$. Alors il existe $c1 = (s, s_1, \dots, s_{k-1}, t)$ et $c2 = (t, t_1, \dots, t_{i-1}, u)$.
Donc en concaténant ces chemins, il existe $c = (s, \dots, s_{k-1}, t, t_1, ;t_{k-1}, u)$, d'où $s * u$.

Ainsi comme les composantes connexes de G sont les classes d'équivalence de $*$, elles forment une partition de S_n.

Q8. Soit s, t deux sommets tels que $s * t$, en notant $len(c)$ la longueur d'un chemin c,
alors $L = \{len(c) | c \text{ chemin de } s \text{ à } t\}$ est une partie de $\mathbb{N}$, non-vide (puisque $s * t$),
donc il existe un plus petit élément k_0 de L. D'où l'existence d'un plus court chemin de s à t.

Soit c_0 un plus court chemin de s à t, notons le $c_0 = (s, s_1, \dots, s_{k_0-1}, t)$.
Si il existe $i \neq j$ tels que $s_i = s_j$ (on peut supposer sans perte de généralité que $i < j$) alors $c = (s, \dots, s_i = s_j, s_{j+1}, \dots, t)$, un chemin de longueur $k_0 - (j - i) < k_0$, ce qui contredit le caractère de plus court chemin de $c_0 \to$ absurde. Donc les sommets d'un plus court chemin sont distincts.

Q9. Soit $k \in [\![0, m[\![$, notons s, t les extrémités de a_k.

Supposons que s et t appartiennent à la même composante connexe de G_k, alors $s *_k t$. Ainsi en notant $c_k = (s_0 = s, s_1, \dots, s_{i-1}, s_i = t)$ (avec $i > 1$), où les sommets de c_k sont adjacents dans G_k, alors il existe un chemin c dans G tel que $c = (s, \dots, t, s)$ (car a_k relie s et t). Or $len(c) = i + 1 \geqslant 2$, donc il existe un cycle dans G. Or G est un arbre donc est acyclique $\to$ absurde !

Ainsi les extrémités de a_k appartiennent à deux composantes connexes différentes de G_k.

En notant pout tout $i \in [\![0, m]\!]$, $\varphi(i)$ le nombre de composantes connexes de G_i, alors $\phi(0) = n$ (G_0 est composé de n sommets non reliés) et $\varphi(m) = 1$ ($G_m = G$ est un arbre, donc connexe).

Donc si $k \in [\![0, m[\![$ et $a_k = \{s, t\}$, alors d'après ce qu'on a fait juste avant, s et t sont dans deux composantes connexes différentes de G_k et dans la même dans G_{k+1}. Les composantes connexes étant disjointes, si $u \in S_n$ tel que $C_{u_k} \neq C_{s_k}$ et $C_{u_k} \neq C_{t_k}$, alors $C_{u_k} = C_{u_{k+1}}$, (les C_{i_k} étant les composantes connexes de G_k contenant i), d'où finalement $\varphi(k+1) = \varphi(k) - 1$

Par une récurrence immédiate, $\varphi(m) = \varphi(0) - m$, d'où $m = n - 1$ et donc le résultat.

Q10. D'après **Q9.**, $(i) \implies (ii)$ et $(i) \implies (iii)$

Si (ii), notons $\mathcal{H} = \{H \mid H = (S_n, B), B \subset A \text{ et H connexe}\}$,

Comme $G \in \mathcal{H}$, alors $\mathcal{H}$ non vide, en particulier, l'ensemble des cardinaux des aretes est une partie non-vide de $\mathbb{N}$, on peut donc considérer $H = (S_n, B)$ avec B de cardinal minimal.

Supposons par l'absurde que H possède un cycle, alors en supprimant une arete quelconque de ce cycle, H reste connexe et possède $|B| - 1 < |B|$ aretes $\to$ absurde car B est de cardinal minimal. Donc H est acyclique et finalement H est un arbre, d'où $|B| = n-1$ (d'après **Q9.**) et donc finalement $H = G$.

Ainsi G est un arbre donc $(ii) \implies (i)$

Si (iii), notons de même $\mathcal{H} = \{H \mid H = (S_n, B), A \subset B$ et H acyclique$\}$,

Comme $G \in \mathcal{H}$, alors $\mathcal{H}$ non vide, alors on peut de même considérer $H = (S_n, B)$ de avec $|B|$ minimal.

Supposons par l'absurde que H ne soit pas connexe, alors il existe $C_1, \dots, C_p$ $(p \geqslant 2)$ composantes connexes de H. Notons n_i et m_i le nombres de sommets et d'aretes de C_i. Alors comme les C_i sont connexes et acycliques, ce sont des arbres, donc possèdent $m_i = n_i - 1$ aretes d'après **Q9.**.

Donc $|B| = \sum_{k=1}^{p} m_k = \sum_{k=1}^{p} n_i - 1 = n - p < n - 1 \rightarrow$ absurde car $|B| \geqslant n - 1$. Donc finalement H est connexe et acyclique, donc est un arbre, donc possède $|B| = n - 1$ aretes.

Donc finalement $H = G$ et donc G est un arbre, d'où $(iii) \implies (i)$

Q11.
```
let rec representant partition sommet =
    if partition.(sommet) < 0 then sommet
    else representant partition partition.(sommet)
```

Q12.
```
let union partition sommet1 sommet2 =
    let h_sommet1 = - partition.(sommet1) - 1 in
    let h_sommet2 = - partition.(sommet2) - 1 in
    if h_sommet1 > h_sommet2 then
        partition.(sommet2) <- sommet1
    else if h_sommet1 = h_sommet2 then
        begin
        partition.(sommet1) <- sommet2 ;
        partition.(sommet2) <- partition.(sommet2) - 1
        end
    else
        partition.(sommet1) <- sommet2 ;;
```

Q13. Montrons le résultat sur k le nombre de réunions réalisés:

Notons H_k : "Après k réunions, si s représentant de $X \in \mathscr{P}$, alors $|X| \geq 2^{h(s)}$"

- Si $k = 0$, alors $\mathscr{P} = \mathscr{P}_n^{(0)}$, donc si $X \in \mathscr{P}$, $X = \{s\}$ et $h(s) = 0$ ainsi $|X| = 1 \geq 2^{h(s)}$, d 'où H_0
- Soit $k \in \mathbb{N}^*$, supposons H_{k-1}, montrons H_k.
  Si $\mathscr{P} = \{X_1, \dots, X_p\}$ une partition ayant subi $k - 1$ réunions depuis $\mathscr{P}_n^{(0)}$.
  Soit $\mathscr{P}'$ ayant subi une réunion depuis $\mathscr{P}$. Notons $\mathscr{P}' = \{X'_1, \dots, X'_{p-1}\}$.

  Soit $i \in [\![0, p[\![$, alors si X'_i n'a pas subi de réunion, alors il existe j tel que $X'_i = X_j$ et donc si s est un représentant de X'_i, alors $h'(s) = h(s)$ et donc $|X'_i| = |X_j| \geqslant 2^{h(s)} = 2^{h'(s)}$.

  Sinon, X'_i a subi une réunion et donc il existe $j \neq m$ tels que $X'_i = X_j \cup X_m$.
  Notons s, s_j, s_m les représentants respectifs de X'_i, X_j, X_m. Lors de la réunion, la hauteur de s ne peut qu'augmenter de 1, donc $h'(s) \leqslant 1 + h(s_j)$ et $h'(s) \leqslant 1 + h(s_m)$.
  Donc, $\mathscr{P}$ étant une partition, $X_j \cap X_m = \varnothing$ et donc,

$$\begin{aligned}|X'_i| &= |X_j| + |X_m| \\ &\geqslant 2^{h(s_j)} + 2^{h(s_m)} \\ &\geqslant 2^{h(s)-1} + 2^{h(s)-1} \\ &= 2^{h(s)}\end{aligned}$$

  D'où H_k, et donc le résultat.

Q14. Si $\mathscr{P}$ est une partition construite depuis $\mathscr{P}_n^{(0)}$, alors dans le pire des cas, trouver le représentant s de $x \in S_n$ se fait en $h(s)$ appels récursifs. Or d'après **Q13.**, $h(s) \leqslant \log_2 |X|$, et $|X| \leqslant n$ (dans le pire des cas c'est une égalité). Donc la complexité de représentant est $\mathcal{O}(\log_2 n)$

La fonction union ne fait que des opérations élémentaires sur des `array` donc est en $\mathcal{O}(1)$

Q15. Pour vérifier que G est un arbre, on vérifie qu'il possède $n - 1$ aretes et qu'il est connexe.

```ocaml
let count_composantes_connexes  graphe =
    let n = Array.length graphe in
    let count =  ref 0 in
    let partition = Array.make n (-1) in
    for sommet1 = 0 to n - 1 do
        List.iter (fun sommet2 ->
                let representant_sommet1 = representant partition sommet1 in
                let representant_sommet2 = representant partition sommet2 in
                if representant_sommet1 <> representant_sommet2 then
                    union partition representant_sommet1 representant_sommet2)
                graphe.(sommet1)
    done;
    for sommet = 0 to n - 1 do
        if partition.(sommet) < 0
            then incr count
    done;
    !count ;;

let est_arbre graphe =
    let n = Array.length graphe in
    (nombre_aretes graphe = n - 1) && (count_composantes_connexes graphe = 1)
```

Q16. Il correspond au chemin $1-2-5-4$

Q17. L'algorithme ne termine pas toujours, en effet si $G = G_{3,2}$ et $\mathcal{T} = (\{0\}, \varnothing)$, alors si $s = 5$, il se peut que l'algorithme fasse le chemin $1-2-6-5$ en boucle, le choix étant aléatoire, et donc l'extrémité d'un tel chemin ne se trouvera jamais dans $\mathcal{T}$.

Q18.

```ocaml
let marche_aleatoire adj parent sommet =
    let chemin = {debut = sommet ; fin = sommet ;
    suivant = Array.make (Array.length adj) 0} in
    while parent.(chemin.fin) = -2 do
        begin
        let nombre_voisins = Array.length adj.(chemin.fin) in
        let indice_voisin_aleatoire = Random.int nombre_voisins in
        let voisin_aleatoire = adj.(chemin.fin).(indice_voisin_aleatoire) in
        chemin.suivant.(chemin.fin) <- voisin_aleatoire; (*si u est dans le cycle,
           cette modification n'importe pas*)
        chemin.fin <- voisin_aleatoire;
        end
    done;
    chemin ;;
```

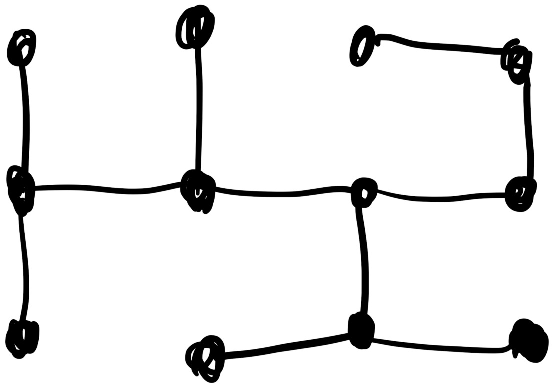
Q19.

```ocaml
let greffe parent chemin =
    let sommet = ref chemin.debut in
    while !sommet <> chemin.fin do (*chemin.fin etant dans T, on a pas à
    ↪  l'ajouté*)
        let suivant = chemin.suivant.(!sommet) in
        parent.(!sommet) <- suivant;
        sommet := suivant
    done;
```

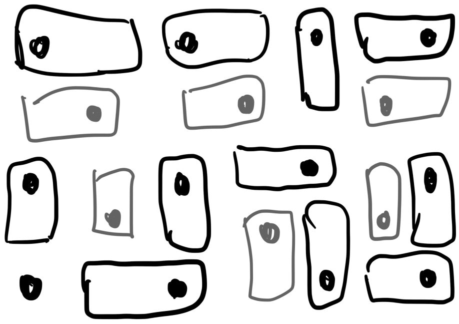
Q20.

```ocaml
let wilson g r =
    let n = Array.length g in
    let adj = adjacence g in
    let parent = Array.make n (-2) in
    parent.(r) <- -1 ;
    for sommet = 0 to n - 1 do
        if parent.(sommet) < 0 then
            let chemin = marche_aleatoire adj parent sommet in
            greffe parent chemin
    done;
    parent ;;
```

Q21.



Q22.



Q23. Si s un sommet de $\mathcal{T}$, alors les coordonnées de s dans $G_{p,q}$ sont $(i = \lfloor s/p \rfloor, j = s \mod p)$.
Comme $G_{p,q}$ ne garde que les cases noires, les coordonnées correspondantes dans $E_{p,q}$ sont $(2i, 2j)$.
Ainsi en fonction de la direction du domino dans la case noire $(2i, 2j)$, on obtient les coordonnées de s', le père de s :

- Si la direction est OUEST, alors $s' = ip + (j-1)$
- Si la direction est NORD, alors $s' = (i+1)p + j$
- Si la direction est SUD, alors $s' = (i-1)p + j$
- Si la direction est EST, alors $s' = ip + (j+1)$

Q24.
```
let coord_noire sommet =
    let i, j = sommet / p, sommet mod p in
    (i * 2, j * 2)
```

Q25.
```
let sommet_direction sommet direction =
    let i, j = sommet / p, sommet mod p in
    match direction with
    |N -> if i >= (q - 1) then -1 else (i + 1) * p + j
    |S -> if i <= 0 then -1 else (i - 1) * p + j
    |W -> if j <= 0 then -1 else i * p + j - 1
    |E -> if j >= (p - 1) then -1 else i * p + j + 1
```

Q26.
```
let phi pavage =
    let parent = Array.make (p * q) (-2) in
    parent.(0) <- -1;
    for i = 0 to q - 1 do
        for j = 0 to p - 1 do
            let sommet = i * p + j in
            let x_pavage, y_pavage = coord_noire sommet in
            if x_pavage mod 2 = y_pavage mod 2
                && (x_pavage, y_pavage) <> (0, 0) then
                let pere = sommet_direction sommet
                                pavage.(x_pavage).(y_pavage) in
                parent.(sommet) <- pere
        done;
    done;
    parent ;;
```