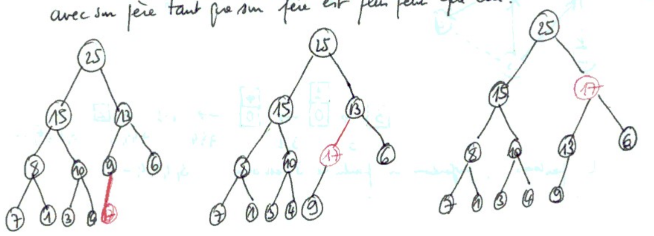
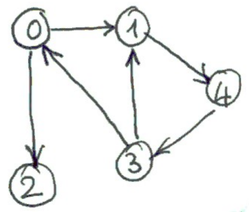
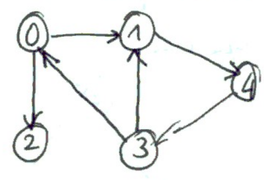
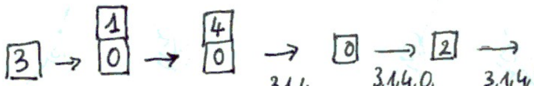
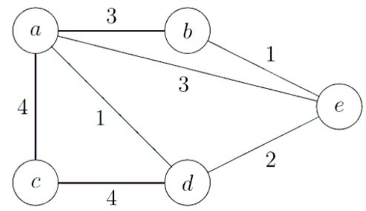
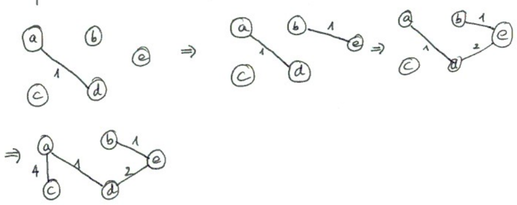
Barème

<table>
<tr><th>Question</th><th>Barème</th></tr>
<tr><td>1. Donner la définition d'un tas-max.</td><td>C.1 : 1|2
0,5 pour complet à gauche et 0,5 pour cds sur étiquettes</td></tr>
<tr><td>2. Donner la représentation sous forme d'arbre du tas-max de représenté par le tableau <code>t=[|25;15;13;8;10;9;6;7;1;3;4|]</code> (on rappelle que le fils gauche de l'élément d'indice i est l'élément d'indice $2i+1$ et son fils droit celui d'indice $2i+2$).</td><td>C.2 : 0.5</td></tr>
<tr><td>3. Décrire un algorithme permettant d'insérer un élément dans un tas-max et le mettre en œuvre pour insérer 17 dans le tas précédent.

Algorithme d'insertion d'un élt dans un tas-max : on place l'élément à insérer au 1er emplacement disponible dans le dernier niveau puis on l'échange avec son père tant que son père est plus petit que lui.
</td><td>C.3 : 1|2
0.5 pour desciption de l'algorithme; 0,5 pour mise en oeuvre sur l'exemple</td></tr>
<tr><td>4. Indiquer la complexité dans le pire des cas de l'opération d'insertion d'un nœud dans un tas de taille n.</td><td>C.4 : 0.5</td></tr>
<tr><td>5. Indiquer à quelle condition un graphe non orienté $G=(S,A)$ est dit connexe.</td><td>C.5 : 1|2</td></tr>
<tr><td>6. Indiquer à quelle condition un graphe non orienté est appelé « arbre ».</td><td>C.6 : 0.5</td></tr>
<tr><td>7.

Parcours en largeur à partir du sommet 3
<table>
<tr><th>Déjà vus</th><th>À traiter.</th></tr>
<tr><td></td><td>3</td></tr>
<tr><td>3</td><td>0, 1</td></tr>
<tr><td>3, 0</td><td>1, 2</td></tr>
<tr><td>3, 0, 1</td><td>2, 4.</td></tr>
<tr><td>3, 0, 1, 2</td><td>4</td></tr>
<tr><td>3, 0, 1, 2, 4.</td><td></td></tr>
</table>
Le parcours en largeur est donc 3, 0, 1, 2, 4.</td><td>C.7 : 1|2
0.5 pour états de A traiter; 0.5 pour parcours</td></tr>
<tr><td>8.

Parcours en profondeur à partir de 3.
Indiquons les états successifs de la Pile "À traiter".

3 → 3, 1 → 3, 1, 4. → 3, 1, 4, 0 → 3, 1, 4, 0, 2
Le parcours en profondeur à partir de 3 est donc 3, 1, 4, 0, 2</td><td>C.8 : 1|2
0.5 pour états de A traiter; 0.5 pour parcours</td></tr>
</table>



On applique par exemple l'algorithme de Kruskal
- on part du graphe comportant tous les sommets de G_2 mais aucune arête
- tant que c'est possible, on rajoute une arête de poids minimum qui relie 2 composantes connexes distinctes du graphe actuel.



C.9 : 1|2
0.5 pour états de A traiter;
0.5 pour parcours

I – **Détermination des voisins des sommets**

Q1 – Il s'agit dans cette question de programmer une fonction qui insère une donnée entière dans une suite triée d'entiers distincts à condition que cette nouvelle donnée ne figure pas déjà dans la suite.

Ocaml : Écrire une fonction `insere` telle que, si `l` est une liste d'entiers distincts triée par ordre croissant et `s` un entie quelconque, `insere l s` renvoie :
- la liste d'entiers obtenue en insérant `s` dans `l` selon l'ordre croissant si la valeur `s` ne figurait pas dans `l`
- la liste `l` si `s` figurait déjà dans `l`

```
let rec insere l s = match l with
   |[] -> [s]
   |t::q when  s < t -> s::l
   |t::q when s = t -> l
   |t::q -> t::(insere q s) ;;
```

P.01 : 2|4

---

2. Le pire des cas se produit lorsque s majore la liste L. Dans ce cas, on effectue $O(|L|)$ opérations :: (où $|L|$ désigne la longueur de L).

P.02 : 1.5|3

Q3 – Il s'agit dans cette question d'écrire une fonction qui donne la liste triée des voisins d'un sommet donné dans un graphe donné.

Ocaml : Écrire une fonction `voisins` telle que `voisins g s` renvoie la liste triée par ordre croissant des voisins du sommet s dans le graphe G. On utilisera pour cela la fonction `insere`.

Version récursive

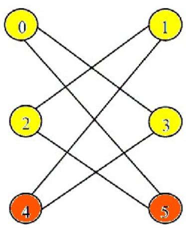
```
let voisins g s =
  let rec aux accu arliste = match arliste with
    |[] -> accu
    |t::q when t.a = s -> aux (insere accu t.b) q
    |t::q when t.b = s -> aux (insere accu t.a) q
    |_::q -> aux accu q
 in aux [] g.al ;;
```

Version itérative

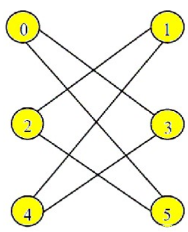
```
let voisins g s =
 let n = List.length g.al  and res = ref [] and l = ref g.al in
 for k = 0 to n-1 do
   let ar = List.hd(!l) in
    if ar.a = s then res := insere (!res) ar.b
    else if ar.b = s then res := insere (!res) ar.a;
    l:=List.tl(!l)
 done;
 !res;;
```

P.03 : 3|3

Q4 – Que donne cet algorithme pour le graphe *Gex2* ci-contre ? Y a-t-il une autre bonne coloration de *G* utilisant moins de couleurs ?



mais



convient aussi

P.04 : 2|2

Q5 – Il s'agit d'écrire en langage de programmation l'algorithme de bonne coloration décrit plus haut. Pour déterminer cette bonne coloration, on utilisera $n(G)$ cases à valeurs entières d'un tableau dont les indices $0, 1, \ldots, n(G)-1$ correspondront aux sommets de G ; la case d'indice s contiendra après le déroulement de l'algorithme la couleur du sommet s.

Ocaml : Écrire en OCaml une fonction `coloration` telle que `coloration g` renvoie le tableau des couleurs attribuées aux sommets du graphe G par l'algorithme décrit ci-dessus.

```
let rec plus_petits  i l = match l with
  |t::q when t < i -> t::(plus_petits i q)
  |_ -> [] ;;
```

retourne la sous-liste de l des éléments < i

```
let rec premier_choix i l = match l with
   |q  when List.mem i q ->  premier_choix (i+1) q
   |_ -> i;;
```

retourne le plus petit entier >= i qui n'est pas dans l

```
let rec applique f l = match l with
   |[] -> []
   |t::q -> (f t)::(applique f q);;

let coloration g = let couleurs = Array.make g.n 0 in
   for i=0 to g.n - 1 do
      let vois = voisins g i in
         let v = plus_petits i vois in
            let f x = couleurs.(x) in
               let c = applique f v in
                  couleurs.(i) <- premier_choix 1 c
   done ;
couleurs ;;
```

P.05 : 3|3

III – Définition du nombre chromatique de *G*

Soit G un graphe et p un entier strictement positif. On appelle **bonne *p*-coloration** de G une bonne coloration n'utilisant que des couleurs appartenant à l'ensemble $\{1, \ldots, p\}$. On introduit les notations suivantes :

- $BC(G, p)$ est l'ensemble des bonnes p-colorations de G ;
- $fc(G, p)$ est le cardinal de $BC(G, p)$;
- $EC(G) = \{p \in \mathbb{N}^* \text{ tel que } fc(G, p) \neq 0\}$.

Q6 – Montrer l'existence d'un unique entier $nbc(G)$ tel que :

$$EC(G) = \{p \in \mathbb{N}^* \text{ tel que } p \geq nbc(G)\}.$$

On dit que $nbc(G)$ est le **nombre chromatique** du graphe G : c'est le nombre minimum de couleurs permettant de colorier les sommets de G de sorte que deux sommets voisins n'aient pas la même couleur.

P.06 : 1.5|2
0.75 pour EC(G) non vide; 0,75 pour EC(G) intervalle

Q7 – Soit G un graphe n'ayant aucune arête. Déterminer $nbc(G)$ en fonction de $n(G)$ et, pour tout $p \in \mathbb{N}^*$, déterminer $fc(G, p)$ en fonction de $n(G)$ et p.

$\boxed{\text{nbc}(G) = 1}$ puisqu'on peut donner la même couleur à tous les sommets.

$f_c(G, p)$ est le nombre de façon d'affecter arbitrairement une des couleurs à chacun des sommets, donc le nombre d'applications de $[|1, n(G)|]$ dans $[|1, p|]$ soit $\boxed{p^{n(G)}}$.

P.07 : 1.5|3
0,5 + 1

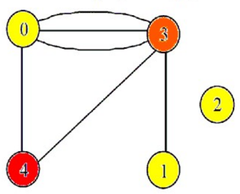
Q8 – Soit G un graphe tel que toute paire de sommets soit une arête. Déterminer $nbc(G)$ en fonction de $n(G)$ et, pour tout $p \in \mathbb{N}^*$, déterminer $fc(G, p)$ en fonction de $n(G)$ et p.

$\boxed{\text{nbc}(G) = n(G)}$ car les couleurs des sommets doivent être deux à deux distinctes et pour $\boxed{p < n(G),\ f_c(G, p) = 0}$.

$\boxed{\text{Pour } p \geq n(G) :\ f_c(G, p) = p(p-1)\ldots(p-n(G)+1) = \dfrac{p!}{(p-n(G))!}}$ car on choisit la couleur du premier sommet (p choix) puis celle du deuxième parmi les $p-1$ couleurs restantes ...

P.08 : 1.5|3
0,5 + 1

Q9 – On considère le graphe *Gex1* de l'exemple introductif. Déterminer $nbc(Gex1)$ et, pour tout $p \in \mathbb{N}^*$, déterminer $fc(Gex1, p)$ en fonction de p.



$\boxed{\text{nbc}(Gex_1) = 3}$ car 0, 3 et 4 doivent avoir trois couleurs différentes, et 1 et 2 peuvent être de la couleur de 0.

Pour $p < 3$: $f_c(G, p) = 0$.

Pour $p \geq 3$: 3 et 2 peuvent être de n'importe quelle couleur : p choix chacun. 0 et 1 doivent être d'une couleur différente de celle de 3 : $p-1$ choix. 4 est de toute couleur sauf celles de 0 et 3 : $p-2$ choix.

Ainsi : $\boxed{f_c(G, p) = p^2(p-1)^2(p-2)}$ pour $p \geq 3$ et cette formule reste valable pour $p < 3$.

P.09 : 1.5|3
0,5 + 1

IV – Les applications *H* et *K*

On dit qu'un sommet d'un graphe est **isolé** s'il n'a aucun voisin. Par exemple, le sommet 2 est isolé dans le graphe *Gex1*.

Q10 – Étant donnés un graphe G et un sommet de G non isolé s, on note $prem\_voisin(G, s)$ le plus petit voisin de s dans G, c'est-à-dire le voisin de s qui a le plus petit numéro. Par exemple, $prem\_voisin(Gex1, 0)$ est le sommet 3.

Ocaml : Écrire en OCaml une fonction `prem_voisin` telle que `prem_voisin g s` renvoie *prem_voisin(G,s)*. Cette fonction ne prévoira pas le cas où s est un sommet isolé de G.

```
let prem_voisin g s = List.hd (voisins g s) ;;
```

puisque `voisins g s` est rangé par ordre croissant.

P.10 : 1

Q11 – On considère un graphe G possédant au moins une arête. On note *prem_ni*(G) le plus petit sommet non isolé du graphe G, c'est-à-dire le sommet non isolé qui a le plus petit numéro. Par exemple, *prem_ni*(*Gex1*) est le sommet 0.

Ocaml : Écrire en OCaml une fonction `prem_ni` telle que `prem_ni g` renvoie *prem_ni(G)*. Cette fonction ne prévoira pas le cas où G ne possède aucune arête.

Version récursive

```
let prem_ni g =
   let rec non_isole s = if (voisins g s =[]) then non_isole (s+1) else s
in non_isole 0 ;;
```

Version itérative

```
let prem_ni g =
 let res = ref 0 in
  while voisins g !res = [] do res := !res + 1
 done ;
 !res ;;
```

P.11 : 2|4

Q12 - Rappelons d'abord que dans la liste $A(G)$ des arêtes d'un graphe G, il est possible qu'une même arête soit répétée.

Étant donné un graphe G possédant au moins une arête, on pose :

$s_1 = prem\_ni(G)$

$s_2 = prem\_voisin(G, prem\_ni(G))$.

Par exemple, pour le graphe *Gex1*, $s_1 = 0$ et $s_2 = 3$.

On note $H(G)$ le graphe obtenu à partir de G en supprimant toutes les arêtes entre s_1 et s_2.

```
let rec filtre p = function
  |[] -> []
  |t::q when p t -> filtre p q
  |t::q -> t :: (filtre p q) ;;
```

retire de la liste argument tous les éléments vérifiant le prédicat p

```
let h g = let s1 = prem_ni g in let s2 = prem_voisin g s1 in
   let f {a = x ; b = y} = (x=s1 && y=s2) || (x=s2 && y=s1) in
     let aprime = filtre f g.al in {n = g.n ; al = aprime} ;;
```

P.12 : 2|4

Q13 – On considère un graphe G possédant au moins une arête. On définit s_1 et s_2 comme dans la question précédente ; on peut remarquer la relation : $s_2 > s_1$. On construit alors un graphe noté $K(G)$ de la façon décrite ci-dessous.

- On construit $H(G)$;
- dans $H(G)$, on superpose s_1 et s_2 ; plus précisément,
  - on considère successivement chaque arête de la liste des arêtes de $H(G)$; pour chacune d'entre elles, on renumérote ses extrémités :
    - une extrémité de valeur strictement inférieure à s_2 est inchangée ;
    - une extrémité de valeur s_2 prend la valeur s_1 ;
    - une extrémité de valeur strictement supérieure à s_2 est décrémentée de 1 ;
  - on diminue de 1 le nombre de sommets du graphe.

La fonction auxiliaire `t` renumérote correctement les états.

```
let t s1 s2 = function
    |s when s < s2 -> s
    |s when s = s2 -> s1 ;
    |s -> s-1 ;;

let k g = let s1 = prem_ni g in let s2 = prem_voisin g s1 in
  let f {a = x; b = y} = {a = t s1 s2 x ; b = t s1 s2 y} in
     let gprime = h g in {n = g.n - 1 ; al = applique f gprime.al} ;;
```

P.13 : 2|4