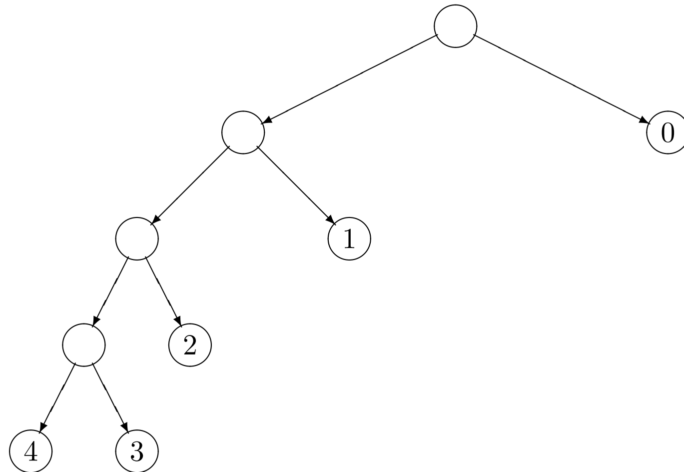
e3a 2017 : option informatique
Un corrigé

Exercice 1

1. Un peigne rangé à cinq feuilles est par exemple :



2. Comme l'énoncé stipule que la hauteur d'une feuille est nulle, il me semble que la définition cohérente de hauteur d'un arbre est le nombre maximal d'arêtes de la racine vers une feuille. Montrons par récurrence sur $n \geq 1$ que la hauteur d'un peigne rangé à n feuilles est égale à $n-1$.
   - Un peigne rangé à 1 feuille es une feuille et sa hauteur vaut 1.
   - Supposons le résultat vrai jusqu'à un rang $n \geq 1$. Un peigne rangé à $n+1$ feuilles s'écrit `Noeud(g,d)` avec `d` qui est une feuille et `g` qui est un peigne rangé à n feuilles. La hauteur de cet arbre est égale à 1 plus le maximum des hauteurs de `g` et `d` qui valent $n-1$ (hypothèse de récurrence) et 1. Notre arbre est donc de hauteur n, ce qu'il fallait prouver.

   Un peigne rangé à n feuilles est de hauteur $n-1$

3. Si l'arbre n'est pas une feuille, on vérifie qu'il y a une feuille à droite et, récursivement, que le fils gauche est un peine rangé.

```
let rec est_range a = match a with
   |Feuille x -> true
   |Noeud (g,Feuille x) -> est_range g
   |_ -> false ;;
```

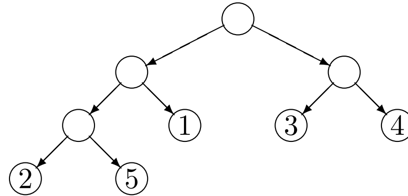
4. Si l'arbre n'est pas une feuille, on vérifie que l'un des fils est une feuille et, récursivement, que l'autre est un peigne strict.

```
let rec est_peigne_strict a = match a with
   |Feuille x -> true
   |Noeud(g,Feuille x) -> est_peigne_strict g
   |Noeud(Feuille x,d) -> est_peigne_strict d
   |_ -> false;;
```

Il reste à particulariser la racine et à vérfiier que, s'ils existent, fils gauche et droit sont des peignes stricts.

```
let est_peigne a = match a with
   |Feuille x -> true
   |Noeud (g,d) -> (est_peigne_strict g)&&(est_peigne_strict d);;
```

5. (a) Une rotation sur l'arbre exemple donne



   (b) La rotation est possible dès que le fils droit est un noeud qui a au moins un fils qui est une feuille.

```
let rotation a = match a with
   |Noeud(a1,Noeud(a2,Feuille f))->Noeud(Noeud(a1,Feuille f),a2)
   |Noeud(a1,Noeud(Feuille f,a2))->Noeud(Noeud(a1,Feuille f),a2)
   |_ -> a;;
```

   (c) Je propose d'écrire une fonction `range` qui agit comme `rangement` mais en SUPPOSANT que l'argument est un peigne. Il reste alors à tester si c'est le cas.
   L'idée de la fonction `range` est d'effectuer une rotation si c'est possible et de recommencer récursivement. On va finir par tomber sur le cas où le fils droit est une feuille. Il suffit alors de ranger le fils gauche.

```
let rec range a = match a with
   |Feuille f -> Feuille f
   |Noeud(g,Feuille f) -> Noeud(range g,Feuille f)
   |_ -> range (rotation a) ;;

let rangement a =
    if est_peigne a then range a
    else a;;
```

Exercice 2

Partie A

1. C'est un parcours classique de liste. Si on sait trouver le minimum de la queue, on sait trouver le minimum de la liste globale.

```
let rec mini l = match l with
   |[x] -> (x,[])
   |x::q -> let (y,r)=mini q in
```

```
        if x<y then (x,y::r)
        else (y,x::r);;
```

2. Notons C_k le nombre de comparaisons effetuées dans l'appel à `mini` avec une liste de taille k. On a $C_1 = 0$ et $C_k = 1 + C_{k-1}$ si $k \geq 2$. On en déduit que

   `mini l` effectue |`l`| comparaisons

3. La question est assez étrange car il ne me semble pas y avoir de tactique "naturelle" utilisant `mini` pour obtenir la liste des absents.
   Dans un premier temps, la fonction `mini` permet de mettre en oeuvre un tri de liste (par selection).

```
let rec tri l = match l with
   |[] -> []
   |_ -> let (x,t)= mini l in
         x::(tri t);;
```

On peut alors écrire une fonction auxiliaire `aux : int list` $\to$ `int` $\to$ `int` $\to$ `list` qui prend en argument une liste `t` supposée triée ainsi que deux entiers i et j. Elle renvoie la liste des éléments $k \in [i, j-1]$ qui ne sont pas dans la liste.

```
let rec aux t i j=
   match t with
  |[] -> if i=j then [] else i::(aux t (i+1) j)
  |x::q -> if i=j then []
           else if i=x then aux q (i+1) j
           else if i<x then i::(aux t (i+1) j)
           else aux q i j;;
```

Il suffit de l'utiliser avec la liste triée ainsi que 0 et n.

```
let absents l n =
    aux (tri l) 0 n;;
```

Il y a d'autres tactiques possibles mais l'énoncé n'est pas assez clair. On pourrait ainsi chercher le minimum m de la liste `l` et ajouter dans une liste en construction les éléments $0, 1, \ldots, m-1$ puis recommencer avec le reste de la liste. On a alors besoin d'une première fonction `liste : int` $\to$ `int` $\to$ `int list` telle que l'appel `liste i j` renvoie la liste des entiers $\geq i$ et $< j$.

```
let rec liste i j =
    if i>=j then []
    else i::(liste (i+1) j);;
```

La fonction `aux : int list` $\to$ `int` $\to$ `int` $\to$ `int list` renvoie la liste des éléments $\geq i$ et $< j$ qui ne sont pas dans la liste.

```
let rec aux l i j = match l with
     |[] -> liste i j
     |_ -> let (m,q)=mini l in
           if m>=j then []
           else (liste i m)@(aux q (m+1) j) ;;
```

Il suffit de l'appeler avec la liste initiale et les entiers 0 et n.

```
let absents l n =
    aux l 0 n;;
```

4. Le tri utilise de l'ordre de n^2 comparaisons du fait des n appels à `mini`. La fonction auxiliaire se contente de parcourir la liste et utilise de l'ordre de n opérations.

`absents l n` effectue $O(n^2)$ comparaisons

Partie B

5. On créé un tableau de bonne taille. On utilise alors une fonction auxilaire `parcours : int list` $\rightarrow$ `unit` qui parcourt la liste en plaçant les élèves.

```
let asseoir l n =
   let tab=make_vect n (-1) in
   let rec parcours l =match l with
      |[] -> ()
      |i::q -> if i<=n || i<0 then tab.(i) <- i ;
               parcours q
   in parcours l;
   tab;;
```

6. La fonction `construit : int` $\rightarrow$ `int list` est telle que dans l'appel `construit i`, on renvoie la liste des absents de numéro $\geq i$.

```
let absent2 l n =
   let tab=asseoir l n in
   let rec construit i =
      if i=n then []
      else if tab.(i) <> -1 then i::(construit (i+1))
      else construit (i+1)
 in construit 0;;
```

7. Créer `tab` coûte de l'ordre de $2n$ opérations (création du tableau et remplissage). La construction de la liste s'opère avec de l'ordre de n consultations du tableau.

`absents2 l n` effectue $O(n)$ accès à une case de tableau

Partie C

8. L'évolution des arguments d'appel est résumé dans le tableau ci-dessous

`tab`	`i`
$[\| -1;4;5;-1;3;0 \|]$	1
$[\| -1;1;5;-1;3;0 \|]$	4
$[\| -1;1;5;-1;4;0 \|]$	3
$[\| -1;4;5;-1;3;0 \|]$	-1

L'appel `place tab i` met l'élève `i` à sa place (en replaçant celui qui était éventuellement en place i, le processus se répétant alors).

9. La fonction remet en ordre dans la classe les élèves présente initialement (un élève i présent initialement sera à la place i en fin de processus).

Exercice 3

Partie A. Calcul des termes de la suite

1. En notant C_n le nombre d'addition induites par l'appel `fibo n`, on a $C_0 = C_1 = 0$ et $\forall n \geq 2$, $C_n = C_{n-1}+C_{n-2}$. Lé résolution de cette récurrence linéaire d'ordre 2 montre que $C_n = O(\phi^n)$ où $\phi = \frac{1+\sqrt{5}}{2}$. La complexité est donc exponentielle vis à vis de n et la fonction est donc peu utile en pratique. Non seulement, le temps d'exécution sera grand mais la pile de récursivité risque d'être vite insuffisante.
2. On gère deux variable `a` et `b` prenant successivement les valeurs (F_0, F_1) puis (F_1, F_2) etc. jusqu'à (F_n, F_{n+1}).

```
let fibo2 n =
   let a=ref 0 and b=ref 1 in
   for i=1 to n do
      let temp= !a in
      a:= !b ;
      b:= !b+temp
      done;
   !a;;
```

Partie B. Décomposition de Zeckendorf

3.

$$20 = 13 + 5 + 2 = F_7 + F_5 + F_3$$

$$21 = F_8$$

$$22 = 21 + 1 = F_8 + F_2$$

4. On imagine un algorithme récurrent pour trouver la décomposition d'un entier n.
   - $1 = F_2$ donne une décomposition de l'entier 1
   - Si $n \geq 2$, on cherche le plus grand indice k tel que $F_k \leq n$ (qui existe et est unique car $(F_p)_{p\geq 1}$ est une suite strictement croissante d'entiers).

     Si $F_k = n$, on a une décomposition

     Sinon $n = F_k + (n - F_k)$ et on sait décomposer $n - F_k$. Le plus grand indice i tel que F_i intervient dans la décomposition de $n - F_k$ n'est pas $k - 1$ car sinon $n - F_k \geq F_{k-1}$ et donc $n \geq F_{k+1}$ ce qui est faux par maximalité de k. On a donc une décomposition valable pour n.

Partie B. Codage de Fibonacci

5. On gère une référence `n` qu'on fait évoluer. Lors du parcours du tableau, on doit ajoputer à `n` une valeur F_i si la case vaut 1. Il serait maladroit de recalculer à chaque fois les F_k. Je gère donc des références `a` et `b` telles que $a = F_{i-1}$ et $b = F_i$ où F_i est le prochain nombre de Fibonacci à utiliser.

```
let decode tab =
   let a=ref 1 and b=ref 1 in
   let n=ref 0 in
   for i=0 to vect_length tab - 1 do
      if tab.(i)=1 then n:= !n+ !b;
      let temp= !a in
```

```
  a:= !b ;
  b:= !b+temp ;
  done;
!n;;
```

6. (a) On suppose connu le codage de $k = b_0 b_1 \dots b_n$ où les b_i valent 0 ou 1 et où deux b_i consécutifs ne peuvent valoir 1. Quitte à ajouter des b_i nuls en fin de chaîne, on peut supposer qu'il existe deux b_i consécutifs nuls. On prend cette première paire $b_i b_{i+1}$.
Le codage peut alors avoir deux formes.

   - Si le codage commence par un 1, il y a une suite de 10 puis au moins un 0 en plus et ainsi
$$k = F_2 + F_4 + \cdots + F_{2p} + b_{2p+3}F_{2p+3} + b_{2p+4}F_{2p+4} + \dots$$
On a alors, comme $1 = F_1$,
$$\begin{aligned} 1+k &= F_1 + F_2 + F_4 + \cdots + F_{2p} + b_{2p+3}F_{2p+3} + b_{2p+4}F_{2p+4} + \dots \\ &= F_3 + F_4 + \cdots + F_{2p} + b_{2p+3}F_{2p+3} + b_{2p+4}F_{2p+4} + \dots \\ &= \dots \\ &= F_{2p+1} + b_{2p+3}F_{2p+3} + b_{2p+4}F_{2p+4} + \dots \end{aligned}$$
et on a une décomposition convenable.
   - Si le codage commence par un 0, on a de même
$$k = F_3 + F_5 + \cdots + F_{2p-1} + b_{2p+2}F_{2p+2} + b_{2p+3}F_{2p+3} + \dots$$
et ainsi, comme $1 = F_2$,
$$\begin{aligned} 1+k &= F_2 + F_3 + F_5 + \cdots + F_{2p-1} + b_{2p+2}F_{2p+2} + b_{2p+3}F_{2p+3} + \dots \\ &= F_4 + F_5 \cdots + F_{2p-1} + b_{2p+2}F_{2p+2} + b_{2p+3}F_{2p+3} + \dots \\ &= \dots \\ &= F_{2p} + b_{2p+2}F_{2p+2} + b_{2p+3}F_{2p+3} + \dots \end{aligned}$$

   L'idée est donc de parcourir le tableau jusqu'à trouver deux 0 consécutifs, à annuler les cases parcourues et à ajouter 1 dans la case où se trouve le premier des deux zéros.
Dans la fonction ci-dessous (non demandée), on suppose le tableau assez grand et on le modifie puis on le renvoie.

```
let plusun tab =
   let i=ref 0 in
   while tab.(!i)==1 || tab.(!i+1)==1 do
        tab.(!i) <- 0 ;
        incr i
        done ;
   tab.(!i) <- 1 ;
   tab;;
```

   (b) Comme on suppose $k \geq 1$, il y a au moins une case du tableau qui vaut 1. Le codage va s'écrire $b_i b_{i+1} \dots$ avec $b_i = 1$ et on a
$$k = F_i + b_{i+1}F_{i+1} + \dots$$

On écrit alors que

$$\begin{aligned} k &= F_{i-2} + F_{i-1} + b_{i+1}F_{i+1} + \dots \\ &= F_{i-4} + F_{i-3} + F_{i-1} + b_{i+1}F_{i+1} + \dots \\ &= F_{i-6} + F_{i-5} + F_{i-3} + F_{i-1} + b_{i+1}F_{i+1} + \dots \\ &= \dots \end{aligned}$$

On cherche donc le premier i tel que `tab.`$(i) = 1$. On met la case à 0 et on repart en arrière en plaçant un 1 une fois sur deux.

```
let moinsun tab=
   let  i=ref 0 in
   while tab.(!i)==0 do incr i done ;
   tab.(!i) <- 0 ;
   decr i ;
   while !i>=0 do
       tab.(!i) <- 1 ;
       decr i;
       decr i
       done ;
   tab;;
```

Exercice 4

ATTENTION : le code informatique ci-dessous n'a PAS été testé. Il est possible qu'il y ait des erreurs (mineures).

1. On gère une liste `l` que l'on va faire grossir. Pour chaque arête `a=[a[0],a[1]]`, on doit ajouter `a[0]` si `a[1]=i` ou `a[1]` si `a[0]=i`.

```
def voisins_G i =
    l=[]
    for a in liste_A:
        if a[0]==i:
            l.append(a[1])
        if a[1]==i:
            l.append(a[0])
    return l
```

2. Les durées étant positives, on peut se contenter de ne regarder que des chemins simples, c'est à dire des chemins qui ne passent pas deux fois par la même station (faire une boucle ne fait qu'augmenter la durée).
   Le graphe étant connexe, il existe au moins un chemin de i vers j. L'ensemble des longueurs de ces chemins (le nombre des arêtes empruntées) est non vide et inclus dans $\mathbb{N}$ et admet donc un minimum. Ce minimum est associé à un chemin simple de vers i vers j.
   L'ensemble des chemins simples de i vers j est non vide. Il est fini (car il n'y a qu'un nombre fini de sommets). L'ensemble des durées des chemins simples de i vers j admet donc un minimum (ensemble non vide et fini).
   Ceci montre qu'il existe un chemin de durée minimum reliant i à j.
3. Si, par l'absurde, il existait un meilleur chemin de i_a à i_b que le sous-chemin $i_a \to \cdots \to i_b$ du chemin considéré alors on pourrait remplacer ce sous-chemin par un autre en faisant diminuer la durée. Ceci contredirait la minimalité du chemin de départ.
   Ceci montre que tout sous-chemin d'un chemin minimal est minimal et entraîne les résultats demandés.

4. Si $(i,j) \in A$, un chemin de i vers j transite forcément par un autre sommet k. En notant k un sommet de transit pour un chemin minimal de i vers j, ce chemin est composé d'un chemin minimal de i vers k puis d'un chemin minimal de k vers j. Son poids est donc $\delta_{min}(i,k) + \delta_{min}(k,j)$. On a donc a fortiori

$$\delta_{min}(i,j) \geq \min_{k \neq i,j} (\delta_{min}(i,k) + \delta_{min}(k,j))$$

Réciproquement, pour un k atteignant le minimum de droite, on construit un chemin de i vers j de poids $\delta_{min}(i,k) + \delta_{min}(k,j)$ en aboutant deux chemins. On a ainsi l'inégalité dans l'autre sens et

$$\forall (i,j) \notin A,\ \delta_{min}(i,j) = \min_{k \neq i,j} (\delta_{min}(i,k) + \delta_{min}(k,j))$$

5. On crée un tableau à deux dimensions avec uniquement des D. On remplit alors les cases comme il faut.

```
Tinit=[[D for i in range(N)] for j in range(N)]
for i in range(N):
    Tinit[i][j]=0
for a in Duree:
    Tinit[a[0]][a[1]]=a[2]
```

6. On crée une copie de `T` et on met à jour ses cases.

```
def FW(T):
    TT=[[T[i][j] for i in range(N)] for j in range(N)]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                TT[i][j]=min(TT[i][j],T[i][k]+T[k][j])
    return TT
```

7. `Tinit` ne prend en compte que les arêtes i.e. les chemins de longueur ≤ 1. Un passage par `FW` permet de prendre en compte les chemins de longueur ≤ 2. Un second passage permet la prise en compte de chemins de longueur ≤ 3. Comme on peut se limiter aux chemins simples de longueur $\leq N-1$, il suffit d'itérer $N-2$ fois.
Dans le code ci-dessous, on modifie `Tinit` à chaque étape.

```
for i in range(N-2):
    Tinit=FW(Tinit)
```

8. Quand on met à jour une case dans `FW`, c'est à dire quand $T'[i,j] < T[i,j]$, on découvre un meilleur chemin allant de i en concaténant des chemins de i à un certain k et de k à j. On pourrait donc gérer un tableau donnant en case (i,j) un chemin optimal de i à j et faire évoluer ce tableau.