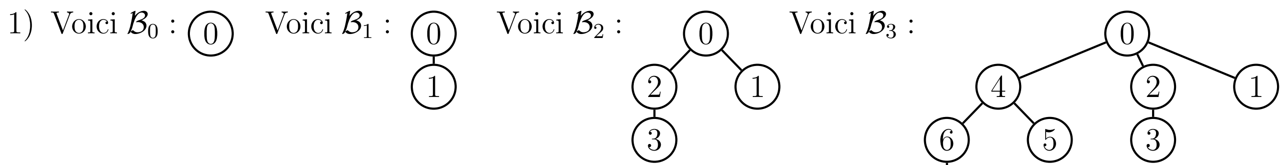
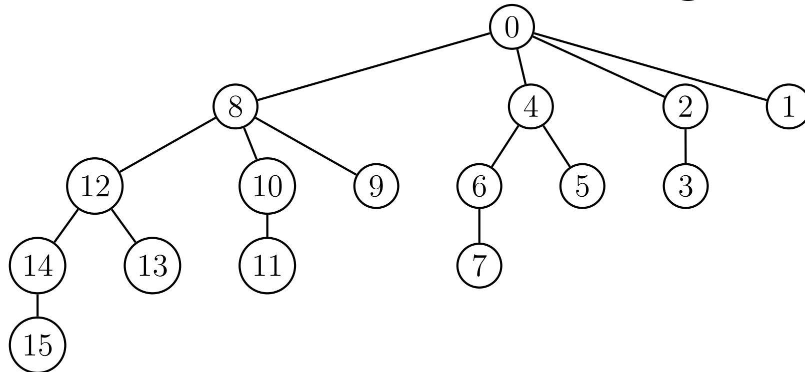
1) Voici $\mathcal{B}_0$: Voici $\mathcal{B}_1$: Voici $\mathcal{B}_2$: Voici $\mathcal{B}_3$:



Voici enfin $\mathcal{B}_4$:



(La numérotation des nœuds est celle qui correspond à l'algorithme de génération des arbres binomiaux donné à la question 5).

2) Soit n_k le nombre de nœuds de $\mathcal{B}_k$. La définition récursive de $\mathcal{B}_k$ donne $n_k = 1 + \sum_{i=0}^{k-1} n_i$ avec comme initialisation $n_0 = 1$.

La récurrence se résout en remarquant que pour tout $n \geqslant 2$, $n_k = \left(1 + \sum_{i=0}^{k-2} n_i\right) + n_{k-1} = 2n_{k-1}$.

Puisque $n_1 = 2$ il vient $\forall k \in \mathbb{N}, n_k = 2^k$.

On aurait aussi pu conjecturer ce résultat depuis le tracé de $\mathcal{B}_4$ et ensuite le prouver par récurrence forte.

Le calcul du nombre e_k de nœuds externes se fait de manière similaire : $e_0 = 1$, $e_1 = 1$ et $\forall k \in \mathbb{N}^*, e_k = \sum_{i=0}^{k-1} e_i$

Ici encore pour $k \geqslant 2$ on a $e_k = \left(1 + \sum_{i=0}^{k-2} e_i\right) + e_{k-1} = 2e_{k-1}$ ce qui donne :

$$e_k = \begin{cases} 1 & \text{si } k = 0 \\ 2^{k-1} & \text{si } k \geqslant 1 \end{cases}$$

3) Soient $\mathcal{B}$ et $\mathcal{B}'$ deux copies de $\mathcal{B}_{k-1}$, soit ℓ la liste des fils de $\mathcal{B}'$ et r' sa racine, alors $\ell = (\mathcal{T}_{k-2} \ldots, \mathcal{T}_0)$ donc $(r', \mathcal{B} :: \ell)$ est un arbre binomial d'ordre k.

4) N'étant pas sûr d'être autorisé à utiliser la fonction `List.map:('a->'b) ->'a list->'b list`, on la code nous même et on la commente ! On obtient le code suivant :

```
(* mapping :  ('a -> 'b) -> 'a list -> 'b list
applique la fonction f a tous les elts de la liste l *)

let rec mapping f l = match l with
  |[] -> []
```

```
  |t::q -> f(t)::(mapping f q)
;;

let rec copie n (Noeud (r, l)) =
  Noeud ((r+n), (mapping (copie n) l))
;;
```

Chaque nœud étant visité une et une seule fois, la complexité est bien proportionnelle à la taille de l'arbre.

5) On utilise les résultats des questions 2 et 3 ainsi que la fonction `copie` pour construire l'arbre binomial d'ordre k. Les noeuds de l'arbre renvoyé par `bin k` auront des étiquettes comprises entre 0 et $2^n - 1$. Afin de construire $\mathcal{B}_k$ on ajoute 2^{k-1} à tous les noeuds d'une copie de $\mathcal{B}_{k-1}$. Pour éviter d'avoir à calculer 2^{k-1} à chaque appel de la fonction, la fonction auxiliaire renvoie aussi la taille de l'arbre.

```
let rec bin k =
   let rec bin_et_taille k =
      if k = 0 then Noeud(0,[]),1
      else let arb,t  = bin_et_taille (k-1) in
      let Noeud(r,l) = a in
      Noeud(r, (copie t a)::l),(2*t)
   in fst(bin_et_taille k);;
```

6) Pour tout $k \in \mathbb{N}^*$, la profondeur $p(k)$ de l'arbre $\mathcal{B}_k$ vérifie la récurrence $p(k) = 1+\max\{p(k-1), \dots, p(0)\}$. On obtient par une récurrence forte immédiate $p(k) = k$.

Soit $k \geq 2$ et $\mathcal{B}_k = (r_k, (\mathcal{T}_{k-1}, \dots \mathcal{T}_0))$. En choisissant des nœuds s_0 et t_0 dans $\mathcal{T}_{k-1}$ et $\mathcal{T}_{k-2}$ de profondeur $k-1$ et $k-2$, `chemin`(s_0, t_0) (dans $\mathcal{B}_k$) est de longueur $(k-1)+1+1+(k-2)$, soit $2k - 1$.

Si maintenant s et t sont deux nœuds de $\mathcal{B}_k$, le chemin $(x_0, x_1, \dots, x_l)$ qui les relie passe par un unique nœud x_i de profondeur minimale. Trois cas sont possibles et pour les étudier, nous notons $p(x, A)$, la profondeur du nœud x dans l'arbre A :

— si $i = 0$, $l = p(x_l, \mathcal{B}_k) - p(x_0, \mathcal{B}_k) \leq p(x_l, \mathcal{B}_k) \leq p(\mathcal{B}_k) = k \leq 2k - 1$;

— si $i = l$, $l = p(x_0, \mathcal{B}_k) - p(x_l, \mathcal{B}_k) \leq p(x_0, \mathcal{B}_k) \leq p(\mathcal{B}_k) = k \leq 2k - 1$;

— sinon, les arbres enracinés en x_{i-1} et en x_{i+1} sont des arbres binomiaux $\mathcal{T}_{k_1}$ et $\mathcal{T}_{k_2}$ d'ordres k_1, k_2 distincts et strictement inférieurs à k. Nous avons alors

$$l = p(x_0, \mathcal{T}_{k_1}) + 1 + 1 + p(x_l, \mathcal{T}_{k_2}) \leq k_1 + 2 + k_2 < (k-1) + 2 + (k-1) = 2k$$

Or la longueur maximale d'un chemin dans $\mathcal{B}_0$ est 0 et dans $\mathcal{B}_1$ est 1, donc pour tout $k \geq 1$, la longueur maximale d'un chemin dans $\mathcal{B}_k$ est $2k - 1$.

7) L'analyse des arbres tracés à la question 1) permet de conjecturer que le nombre de nœuds de profondeur ℓ de l'arbre $\mathcal{B}_k$ est $\binom{k}{\ell}$.

Montrons par récurrence sur k, la propriété $\mathcal{P}_k$: $\forall \ell \in \mathbb{N}$ le nombre de nœuds de profondeur ℓ de l'arbre $\mathcal{B}_k$ est $\binom{k}{\ell}$.

— Initialisation : Dans $\mathcal{B}_0$, il y a un seul nœud de profondeur 0 ($\binom{k}{0} = 1$) et aucun nœud de profondeur strictement positive ($\forall \ell \in \mathbb{N}^*$, $\binom{0}{\ell} = 0$). $\mathcal{P}_0$ est donc vraie.

— Hérédité : Soit $k \in \mathbb{N}^*$, supposons $\mathcal{P}_{k'}$ vraie pour tout $k' \in [\![0, k-1]\!]$ et montrons $\mathcal{P}_k$. Soit $\ell \in \mathbb{N}$.

  — si $\ell = 0$: il y a bien un seul nœud de profondeur 0 dans $\mathcal{B}_k$ ($\binom{k}{0} = 1$).

  — si $\ell > 0$: le nombre de nœuds de profondeur ℓ est le nombre de nœuds de profondeur $\ell - 1$ de chacun des fils, c'est–à–dire $\sum_{0 \leqslant k' \leqslant k-1} \binom{k'}{\ell - 1}$

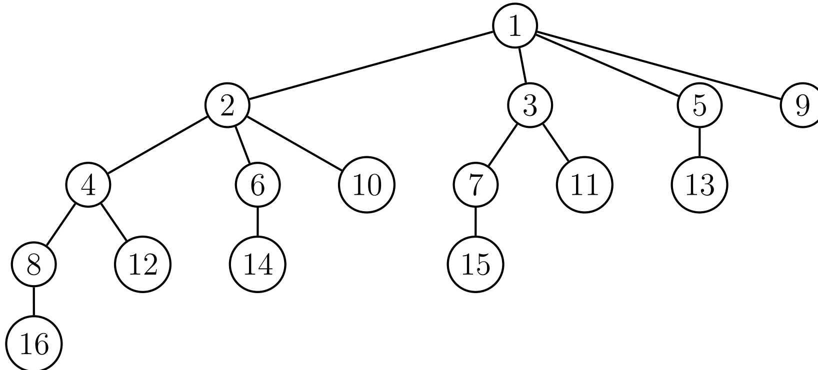
  Appliquons la formule du triangle de Pascal pour faire apparaître un télescope :

$$\sum_{0 \leqslant k' \leqslant k-1} \binom{k'}{\ell-1} = \sum_{k'=0}^{k-1} \left[\binom{k'+1}{\ell} - \binom{k'}{\ell} \right] = \binom{k}{\ell} - \binom{0}{\ell} = \binom{k}{\ell}$$

  $\mathcal{P}_k$ est donc vraie.

— Conclusion : par le principe de récurrence, $\mathcal{P}_k$ est prouvée pour tout $k \in \mathbb{N}$.

8) Voici $\mathcal{C}_{16}$:



Rappelons la définition d'un arbre : c'est un ensemble fini de sommets, muni d'une relation “a pour père”, telle que (dans un arbre non vide) chaque sommet a exactement un père sauf un sommet qu'on appelle la racine, et telle qu'il n'y a pas de cycle.

Il reste donc deux choses à vérifier :

— qu'il s'agit bien d'une relation, c'est–à–dire que $\forall i \in [\![2, n]\!]$, $i - 2^{\lceil \log_2(i) \rceil - 1} \in [\![1, n]\!]$

— qu'il n'y a pas de cycle

Soit $i \geqslant 2$, commençons par prouver que $i - 2^{\lceil \log_2(i) - 1 \rceil - 1} \geqslant 1$.

$\lceil \log_2(i) \rceil < \log_2(i) + 1$ donc $\lceil \log_2(i) \rceil - 1 < \log_2(i)$ donc $2^{\lceil \log_2(i) \rceil - 1} < i$ or ce sont des entiers donc $2^{\lceil \log_2(i) \rceil - 1} \leqslant i - 1$ donc $i - 2^{\lceil \log_2(i) \rceil - 1} \geqslant 1$.

Pour la suite de la preuve, remarquons que $\forall i \in [\![2, n]\!], i - 2^{\lceil \log_2(i) \rceil - 1} < i \leqslant n$ donc on a bien le premier point de la preuve : $\forall i \in [\![2, n]\!], i - 2^{\lceil \log_2(i) \rceil - 1} \in [\![1, n]\!]$.

Par ailleurs, la remarque $i - 2^{\lceil \log_2(i) \rceil - 1} < i$ prouve qu'il n'y a pas de cycle (sinon il y aurait forcément un sommet dont le numéro serait inférieur ou égal à celui de son père).

Enfin, $\mathcal{C}_{16}$ et $\mathcal{B}_4$ sont isomorphes. Il semble raisonnable de conjecturer que pour tout $k \in \mathbb{N}$ $\mathcal{C}_{2^k}$ et $\mathcal{B}_k$ sont isomorphes.

9) Une question difficile. Voici succinctement les arguments qui aboutissent au programme solution de cette question :

Pour tous $\mathtt{i}, \mathtt{j} \in [\![1, \mathtt{n}]\!]$, $\mathtt{j}$ est un fils de $\mathtt{i}$ si et seulement si l'entier $\mathtt{p} = \mathtt{j} - \mathtt{i}$ est une puissance de 2 égale à $2^{\lceil \log_2(\mathtt{i}) \rceil - 1}$ c'est–à–dire supérieure ou égale à $\mathtt{j}/2$.

Ainsi les fils de $\mathtt{i}$ sont exactement les $\mathtt{i} + \mathtt{p}$ où $\mathtt{p}$ est une puissance de 2 telle que $\mathtt{i} \leqslant \mathtt{p}$ et $\mathtt{i} + \mathtt{p} \leqslant \mathtt{n}$.

Encore une remarque : soient $\mathtt{i}, \mathtt{j}, \mathtt{k} \in [\![1, \mathtt{n}]\!]$ tels que $\mathtt{j}$ est un fils de $\mathtt{i}$, et $\mathtt{k}$ est le plus petit fils de $\mathtt{j}$. Soient $\mathtt{p}, \mathtt{q}$ les puissances de 2 telles que $\mathtt{j} = \mathtt{i} + \mathtt{p}$ et $\mathtt{k} = \mathtt{j} + \mathtt{q}$. Alors $\mathtt{i} \leqslant \mathtt{p}$ donc $\mathtt{p} < \mathtt{j} = \mathtt{i} + \mathtt{p} \leqslant 2\mathtt{p}$ donc la plus petite puissance de 2 supérieure ou égale à $\mathtt{j}$ est $\mathtt{q} = 2\mathtt{p}$ ce qui permet de calculer très facilement $\mathtt{q}$.

Ainsi lorsque nous descendons dans l'arbre, nous gardons en mémoire le paramètre $\mathtt{p}$ qui correspond à la puissance de 2 que l'on rajoute à un père pour construire un de ses fils : pour la construction des fils de ses fils il faudra partir de la puissance de 2 égale à $2\mathtt{p}$.

On utilise une fonction auxiliaire récursive :

```
fils_depuis_pere : int -> int -> arbre list
```

telle l'appel de `fils_depuis_pere i p` génère la liste des fils de `i`, en rajoutant à `i` des puissances de 2 successives en partant de `p`.

```
let cn n =
  let rec fils_depuis_pere i p =
    let (j, q) = (i+p, 2*p) in
    if j > n then []
    else (Noeud (j, fils_depuis_pere j q))::(fils_depuis_pere i q)
  in
  Noeud (1, fils_depuis_pere 1 1)
;;
```

10) On utilise une fonction `max_liste : int list -> int` qui calcule l'élément maximum d'une liste d'entiers. D'où le code :

```
let rec max_liste = function
  | [] -> -1
  | x::q -> max x (max_liste q)
;;

let rec profondeur = function (Noeud (_, l)) ->
    1 + max_liste (mapping profondeur l)
;;
```

11) Pour ne visiter qu'une seule fois chaque nœud on utilise une fonction auxiliaire récursive `aux : arbre -> int * int` qui retourne un couple formé du numéro d'un nœud externe de profondeur maximum, mais aussi de la profondeur de ce nœud.

Pour programmer cette fonction on utilise la même méthode que pour le calcul de la profondeur, sauf qu'ici on utilise une fonction :

`max_couple_ordonnee : int * int list -> int * int` qui calcule dans une liste de couple l'élément dont la seconde coordonnée est maximum.

```
let rec max_couple_ordonnee = function
```

```
  | [] -> (0, 0)
  | (x, y)::q -> let (a, b) = max_couple_ordonnee q in
                 if b > y then (a, b) else (x, y)
;;

let noeud_externe_max a =
  let rec aux = function
    | Noeud (r, []) -> (r, 0)
    | Noeud (_, l) ->
      let (x, y) = max_couple_ordonnee (mapping aux l) in (x, y+1)
  in
  fst (aux a)
;;
```

12) Voici une première solution sans mécanisme d'exception :
on remonte un couple (`trouve`, `chemin`) : `bool` $*$ `liste`, le booléen indiquant si la recherche passant par ce nœud aboutit bien, et dans ce cas, le second membre du couple est le chemin à suivre. On utilise deux fonctions auxiliaires récursives croisées :

```
descend_noeud : liste -> arbre -> bool * liste
descend_liste : liste -> arbre list -> bool * liste
```

La première consiste à descendre depuis un nœud, la seconde depuis une liste de nœuds. Dans les deux cas le premier paramètre est le chemin parcouru jusque là.

```
let chemin a s =
  let rec descend_noeud (ch:liste) = function Noeud (x, fils) ->
    if x = s then (true, x::ch)
    else descend_liste (x::ch) fils
  and descend_liste (ch:liste) = function
    | [] -> (false, [])
    | x::q -> let (u, v) = descend_noeud ch x in
              if u then (u, v)
              else descend_liste ch q
  in
  List.rev (snd (descend_noeud [] a))
;;
```

Voici maintenant une seconde solution qui utilise un mécanisme d'exception : on lève une exception dès qu'on a trouvé le nœud `s` et on récupère le chemin.

```
exception Trouve of liste;;

let chemin a s =
  let rec descend ch = function Noeud (x, f) ->
    if x = s then raise (Trouve (x::ch))
    else List.map (descend (x::ch)) f;
    []
```

```
    in
    try descend [] a
    with Trouve ch -> List.rev ch
  ;;
```

13) Commençons par écrire une fonction `un_pivot : arbre -> int -> arbre` qui effectue cette opération mais uniquement lorsque le nœud passé en second paramètre est un fils de la racine. Dans le code du programme, ce fils de la racine qui deviendra la nouvelle racine s'appelle "fiston".

On utilise pour cela une fonction auxiliaire récursive `retire_fiston : arbre list -> arbre list * arbre list` qui calcule deux listes : la liste des fils du fiston, et la liste des fils de la racine privée du fiston.

```
let un_pivot (Noeud (racine, lfils)) fiston =
  let rec retire_fiston = function
    | [] -> failwith "Impossible : le fiston est absent"
    | (Noeud (y, ff) as g)::q -> if y = fiston then (ff, q)
      else let (u, v) = retire_fiston q in
           (u, g::v)
  in
  let (u, v) = retire_fiston lfils in
  Noeud (fiston, (Noeud (racine, v))::u)
;;
```

Voici maintenant la fonction principale : on calcule le chemin depuis la racine jusqu'au nœud, et on effectue les pivots un à un le long de ce chemin. Noter que dans le chemin calculé par la fonction `chemin`, il faut retirer le premier élément qui est la racine.

```
let pivot a noeud =
  let rec pivote a = function
    | [] -> a
    | x::q -> pivote (un_pivot a x) q
  in
  match chemin a noeud with
    | [] -> failwith "pas possible"
    | _::q -> pivote a q
;;
```

14) Pour s et t nœuds de $\mathcal{T}$, nous noterons $d(s,t)$ la longueur de `chemin`(s,t). Les chemins étant les mêmes dans $\mathcal{T}$ et dans $\mathcal{T}'$, nous devons démontrer :

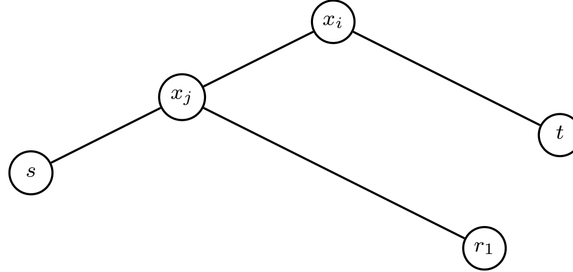
$$\max_{s,t\in\mathcal{N}(\mathcal{T})} d(s,t) = \max_{s\in\mathcal{N}(\mathcal{T})} d(s,r_1)$$

ce qui revient à

$$\forall s,t \in \mathcal{N}(\mathcal{T}),\ \exists s' \in \mathcal{N}(\mathcal{T}),\ d(s,t) \leq d(s',r_1).$$

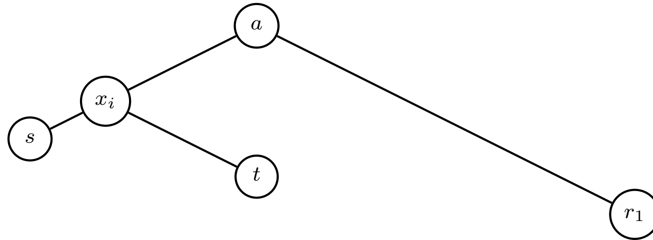
Fixons donc deux nœuds s et t de $\mathcal{T}$ et notons $(x_0,\dots,x_k)$ le chemin de s à t et x_i le nœud le moins profond de ce chemin (x_i est le plus proche ancêtre commun à s et t). Deux cas se présentent :

— Premier cas : x_i est un ancêtre de r_1. x_i est alors ou bien le plus proche ancêtre commun à s et r_1, ou bien le plus proche ancêtre commun à t et r_1. Par symétrie, supposons que x_i est le plus proche ancêtre commun à t et r_1. Cette situation peut se schématiser de la façon suivante :



Comme r_1 est plus profond que s dans l'arbre enraciné en x_i, nous avons $d(t, r_1) = d(t, x_i) + d(x_i, r_1) \geq d(t, x_i) + d(x_i, s) = d(s, t)$.

— Deuxième cas : x_i n'est pas un ancêtre de r_1. En notant a le plus proche ancêtre commun à x_i et r_1, nous obtenons le schéma :



On a cette fois $d(t, r_1) = d(t, a) + d(a, r_1) \geq d(t, a) + d(a, s) \geq d(t, x_i) + 1 + d(x_i, s) = d(s, t)$.

15) Il suffit de combiner les fonctions précédentes :

```
let diametre a = profondeur (pivot a (noeud_externe_max a));;
```

16) Soit r un nœud de $\mathcal{T}$ et $\mathcal{T}' = pivot(\mathcal{T}, r)$. On a clairement :

$$\forall\, s, t \in \mathcal{N}(\mathcal{T}'),\quad \begin{cases} p(s, \mathcal{T}') = d(s, r) \leq D \\ d(s, t) \leq p(s, \mathcal{T}') + p(t, \mathcal{T}') \leq 2p(\mathcal{T}') \end{cases}$$

d'où $p(\mathcal{T}') \leq D \leq 2p(\mathcal{T}')$, i.e. $\left\lceil \dfrac{D}{2} \right\rceil \leq p(\mathcal{T}) \leq D$.

En choisissant $r = r_1$, $p(\mathcal{T}') = D$.

Posons $k = \lceil D/2 \rceil$ et fixons deux sommets s et t de $\mathcal{T}$ tels que $chemin(s, t) = (x_0, x_1, \ldots, x_D)$ soit un chemin de longueur maximale dans $\mathcal{T}$. En choisissant $r = x_k$, l'arbre $\mathcal{T}'$ est de profondeur k. En effet, $p(x_0, \mathcal{T}') = k$ et si s est un nœud de $\mathcal{T}'$, r est le plus proche ancêtre commun ou bien du couple (x_0, s), ou bien du couple (s, x_D).
Dans le premier cas, $D \geq d(x_0, s) = k + p(s, \mathcal{T}')$, puis $p(s, \mathcal{T}') \leq D - k \leq k$.
Dans le second cas, $D \geq d(s, x_D) = p(s, \mathcal{T}') + D - k$, puis $p(s, \mathcal{T}') \leq k$.

Nous en déduisons :

$$\max_{r\in\mathcal{N}(\mathcal{T})} p(pivot(r,\mathcal{T})) = D \text{ et } \min_{r\in\mathcal{N}(\mathcal{T})} p(pivot(r,\mathcal{T})) = \left\lceil \frac{D}{2} \right\rceil .$$

17) Prouvons que la durée de diffusion pour la numérotation naturelle dans l'arbre $\mathcal{B}_k$ est $\dfrac{k(k+1)}{2}$.

Raisonnons par récurrence forte sur k. Pour l'arbre $\mathcal{B}_0$ la durée de diffusion est 0.

Fixons $k \in \mathbb{N}^*$ et supposons le résultat pour chaque entier $i \in [\![0, k-1]\!]$.

Par hypothèse de récurrence, pour chaque fils $\mathcal{B}_i$ (avec $0 \leqslant i \leqslant k-1$) de la racine, le message arrvie à la racine r_{i+1} du fils $\mathcal{B}_i$ à la date $i+1$, donc il est transmis dans tout le fils $\mathcal{B}_i$ à la date $i+1+\dfrac{i(i+1)}{2} = \dfrac{(i+1)(i+2)}{2}$. C'est donc dans le fils $\mathcal{B}_{k-1}$ de la racine que le message est le plus long à être diffusé, la diffusion se termine à la date $\dfrac{k(k+1)}{2}$.

Par récurrence, la propriété est prouvée.

18) Prouvons que la durée de diffusion pour la numérotation renversée dans l'arbre $\mathcal{B}_k$ est k.

Raisonnons par récurrence forte sur k. Pour l'arbre $\mathcal{B}_0$ la durée de diffusion est 0.

Fixons $k \in \mathbb{N}^*$ et supposons le résultat pour chaque entier $i \in [\![0, k-1]\!]$.

Par hypothèse de récurrence, pour chaque fils $\mathcal{B}_i$ (avec $0 \leqslant i \leqslant k-1$) de la racine, le message arrive à la racine r_{i+1} du fils $\mathcal{B}_i$ à la date $k-i$, donc il est transmis dans tout le fils $\mathcal{B}_i$ à la date $k-i+i=k$. La diffusion s'arrête donc en même temps pour chaque fils à la date k.

Par récurrence, la propriété est prouvée.

19) La profondeur de $\mathcal{B}_k$ est la longueur d'un chemin particulier entre sa racine et un nœud externe : c'est donc un minorant de la durée de diffusion optimale. Par ailleurs le résultat de la question 18 montre que la profondeur est la durée de diffusion dans un cas particulier. On obtient donc le fait que la durée de diffusion optimale dans $\mathcal{B}_k$ est k.

20) On utilise un approche récursive : soit a un arbre dont les fils de la racine sont notés $f_1,\dots,f_n$. On appelle récursivement l'algorithme sur $f_1,\dots,f_n$: notons $o_1,\dots,o_n$ les duŕées de diffusion optimales respectivement pour les sous–arbres issus de $f_1,\dots,f_n$.

Quitte à trier la liste $(o_1,\dots,o_n)$ par ordre décroissant on peut supposer $o_1 \geqslant o_2 \geqslant \cdots \geqslant o_n$. On diffuse alors dans les fils par ordre de temps de diffusion décroissant, c'est–à–dire d'abord le sous–arbre issu de f_1, puis le sous–arbre issu de f_2,...enfin le sous–arbre issu de f_n.

Au total le temps de diffusion optimal sera $\max\{o_i + i \,|\, 1 \leqslant i \leqslant n\}$.

Si on admet que le degré de l'arbre (c'est–à–dire le nombre maximum de fils d'un nœud) est borné alors l'opération de tri est en temps constant et chaque nœud est visité une seule fois d'où une complexité proportionnelle à la taille.

21) On code l'algorithme de la question précédente. On utilise une fonction auxiliaire récursive

```
maxi_et_incremente : int -> int -> liste -> int
```

telle que l'appel de `maxi_et_incremente 1 0 l` où `l` est la liste $[\mathtt{l}_0;\dots,\mathtt{l}_{k-1}]$ permet de calculer $\max\{\mathtt{l}_0+1, \mathtt{l}_1+2,\dots,\mathtt{l}_{n-1}+\mathtt{n}\}$. En cas de liste vide la valeur de retour est 0.

Rappelons que la fonction `List.sort (fun x y -> y - x) : liste -> liste` issue de la bibliothèque standard d'OCaml permet de trier une liste par ordre décroissant.

```
let rec diffusion_optimale = function (Noeud (x, f)) ->
  let rec maxi_et_incremente i maxi = function
    | [] -> maxi
    | x::q -> maxi_et_incremente (i+1) (max maxi (x+i)) q in
  maxi_et_incremente 1 0 (List.sort (fun x y -> y - x)
                                (mapping diffusion_optimale f))
;;
```

22) La borne supérieure demandée est $n-1$ où n est la taille de l'arbre. En effet à chaque étape de diffusion il y a au moins un nouveau nœud qui reçoit le message dans au bout d'au plus $n-1$ étapes tous les nœuds auront reçu le message.

Cette borne supérieure est atteinte par exemple dans le cas d'un arbre de nœuds $i_1, \dots, i_n$ où la racine est i_1 et pour tout $k \in [\![1, n-1]\!]$, i_{k+1} a pour père i_k. Dans ce cas chaque nœud possède exactement un fils (sauf i_n), et il y a exactement une stratégie de diffusion possible : dès qu'un nœud reçoit le message, il le transmet à son fils.

23) Prenons le problème dans l'autre sens. On essaye de savoir, au bout de p transmissions le nombre maximal de noeuds ayant reçu le message. Notons α_p ce nombre. Après 0 transmission, il n'y a qu'un seul nœud qui a le message donc $\alpha_0 = 1$, Après une transmission, il y a au maximum deux noeuds ayant reçu le message et donc $\alpha_1 = 2$. A l'étape suivante, les deux noeuds peuvent diffuser l'information et donc $\alpha_2 = \alpha_1 + 2 = 4$. De manière générale, on voit que pour tout entier k, $\alpha_{k+1} = 2.\alpha_k$ car chacun des α_k noeud peut diffuser l'information à un nouveau noeud. On obtient donc que pour tout entier k, $\alpha_k = 2^k$.

La durée minimale de diffusion dans un arbre ayant n noeuds est donc supérieure ou égale à $p = \lceil \log_2(n) \rceil$. En effet, pour tout $k < p$, $2^{p-1} < n \leq 2^p$ et donc pour tout $k < p$, $k \leq p-1$ et donc $2^k < n$. On ne peut donc pas diffuser le message aux n noeuds de l'arbre en k étapes.

Réciproquement, en s'inspirant de ce qui précède, montrons que pour tout entier n, il existe un arbre ayant n noeuds dans lequel on peut diffuser le message aux n noeuds en $p = \lceil \log_2(n) \rceil$ étapes. Il suffit de considérer un sous-arbre $\mathcal{T}$ de $\mathcal{B}_p$ obtenu en supprimant $2^p - n$ feuilles (on sait que $2^{p-1} < n \leq 2^p$ et donc $2^p - n$ est strictement inférieur à 2^{p-1} qui est le nombre de feuilles de l'arbre $\mathcal{B}_p$). Comme on sait que l'on peut diffuser le message dans l'arbre $\mathcal{B}_p$ en p étapes, on peut, à fortiori, diffuser le message dans $\mathcal{T}$ en p étapes.

24) On a vu dans la partie précédente que, dans l'arbre $\mathcal{B}_k$, on peut diffuser un message de la racine vers tous les noeuds en k étapes.

En lisant les échanges dans le sens inverse, on peut faire remonter le message de tous les noeuds à la racine de $\mathcal{B}_k$ en k étapes.

Voici alors l'algorithme. On rappelle que pour $k \geq 1$, l'arbre $\mathcal{B}_k$ peut être vu comme deux arbres $\mathcal{B}_{k-1}$ notés A et B tel que r_B (la racine de B) soit un fils de r_A.

— Tous les noeuds de A remontent leur message à r_A et tous les noeuds de B remontent leur message à r_B. Cela prend $k-1$ étapes.
— Les racines r_A et r_B échangent les messages. A ce moment, les deux racines connaissent tous les messages de l'arbre.

— On reprend la première étape mais dans l'autre sens : la racine r_A diffuse le message total dans l'arbre A et la racine r_B diffuse le message total dans B. Là encore, il y a $k-1$ étapes.

Au total, l'algorithme comporte $(k-1)+1+(k-1) = 2k-1$ étapes.

25) Notons u le nœud externe de profondeur maximum de $\mathcal{B}_k$, il est de profondeur k et il est un nœud du fils $\mathcal{B}_{k-1}$ de la racine. Notons aussi v le fils de profondeur maximum dans le fils $\mathcal{B}_{k-2}$ de la racine de $\mathcal{B}_k$: il est de profondeur $\mathcal{B}_{k-1}$ dans $\mathcal{B}_k$. Puisque u et v appartiennent à deux fils différents de la racine, un chemin de u à v doit passer par la racine, donc ce chemin est de longueur $2k-1$. Donc pour faire passer le message connu de u à v, on a besoin d'une durée supérieure ou égale à $2k-1$.

26) La question est mal posée. Que signifie « **une** borne inférieure » ? On peut penser que l'on attend juste une minoration non triviale (genre 0). On voit qu'après k étapes, le noeud le plus informé détient au maximum 2^k messages. Il est donc nécessaire que le nombre p d'étapes de la transmission vérifie $2^p \geq n$. Donc $p \geq \lfloor \log_2(n) \rfloor$.

On peut aussi minorer par le diamètre de l'arbre.

27) Dans l'étape 1 on efface $n-2$ nœuds ce qui fait $n-2$ échanges. Dans l'étape 2 on effectue un échange. Dans l'étape 3 on fait $n-2$ échange comme dans l'étape 1. Au total ça fait $n-2+1+n-2 = 2n-3$ échanges.

28) Il suffit d'effectuer un parcours de l'arbre (c'est–à–dire une liste de ses sommets) de façon à ce que tout père apparaisse après ses fils. Ensuite on ne garde que les $n-2$ premiers éléments de la liste.

Le programme ci–dessous effectue plutôt un parcours préfixe (c'est–à–dire qu'il met chaque père avant ses fils) puis on renverse la liste. Ça a l'avantage de faciliter l'extraction des deux nœuds surnuméraires que l'on extrait en tête de liste plutôt qu'en queue.

Pour le parcours préfixe, on utilise la méthode usuelle de fonctions récursives croisée, l'une dont le paramètre est un arbre (ou un nœud), l'autre dont le paramètre est une liste d'arbres.

```
let liste_sommets ab =
  let sommets = ref [] in
  let rec parcours_liste listarbre = match listarbre with
    | [] -> ()
    | a1 :: q -> parcours_arbre a1; parcours_liste q
  and
  parcours_arbre (Noeud(x,ll)) =
    parcours_liste ll;
    sommets := x:: !sommets
  in parcours_arbre ab; !sommets;;
```

On peut alors écrire la fonction

```
let echange_total a =
  let l = liste_sommets a in
  match l with
     [] | [_] -> failwith "pas assez de noeuds"
    | x :: y :: q -> miroir q;;
```

où `miroir : 'a list -> 'a list` est une fonction qui renverse une liste.

```
let miroir l =
  let rec aux li acc =
    match li with
      |[] -> acc
      |x::q -> aux q (x::acc)
  in aux l [];;
```

29) Au cours de ces échanges, il existe un premier instant i_0 pour lequel un des nœuds “connaît” tous les messages. Soit $\{a_{i_0}, b_{i_0}\} = \{\alpha, \beta\}$. Ainsi, les nœuds α et β sont les seuls nœuds à posséder toute l'information à l'instant i_0. Si nous supprimons l'arête (α, β) de $\mathcal{T}$, nous séparons le graphe $\mathcal{T}$ en deux arbres $\mathcal{T}_\alpha$ et $\mathcal{T}_\beta$ de taille n_α et n_β

À l'instant $i_0 - 1$, le nœud α connaît toutes les informations des nœuds de $\mathcal{T}_\alpha$: chaque arête de $\mathcal{T}_\alpha$ a donc été utilisée au moins une fois entre l'instant 1 et l'instant $i_0 - 1$ (si γ est un nœud de $\mathcal{T}_\alpha$ distinct de α, l'échange doit nécessairement utiliser l'arête liant γ à son père pour que l'information msg_γ arrive à la racine α). De façon symétrique, c'est également le cas pour les arêtes de $\mathcal{T}_\beta$. Comme le nombre d'arête d'un arbre est égal à son nombre de nœuds diminué de 1, nous avons :

$$i_0 - 1 \geq n_\alpha - 1 + n_\beta - 1 = n - 2$$

À l'instant $i_0 - 1$, le nœud α ne connaît pas au moins une information msg_γ, où γ est un nœud de $\mathcal{T}_\beta$. On en déduit qu'à l'instant i_0, aucun nœud de $\mathcal{T}_\alpha$ autre que α ne connaît l'information msg_γ. Comme précédemment, chaque arête de $\mathcal{T}_\alpha$ est utilisée au moins une fois entre l'instant $i_0 + 1$ et N. De façon symétrique, c'est également le cas des arêtes de $\mathcal{T}_\beta$, ce qui donne :

$$N - i_0 \geq n_\alpha - 1 + n_\beta - 1 = n - 2$$

Nous obtenons ainsi

$$N = i_0 - 1 + 1 + N - i_0 \geq 2n - 3$$