# I Quelques fonctions auxiliaires

1. Attention à ne pas compter chaque arête deux fois.

```
let nombre_aretes g =
    let res = ref 0 in
    for i = 0 to Array.length g - 1 do
        res := !res + List.length g.(i)
    done;
    !res/2 (* on a compté chaque arête 2 fois *)
```

Autre possibilité :

```
let nombre_aretes g =
    (Array.map List.length g
    |> Array.fold_left (+) 0) / 2;;
```

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

3. Une possibilité : `let adjacence g = Array.map Array.of_list g`
Si $n$ est le nombre de sommets et $m$ le nombre d'arêtes de `g`, `Array.map Array.of_list g` crée un nouveau tableau de taille $n$ (complexité $\mathrm{O}(n)$) puis le remplit en appliquant `Array.of_list` sur chaque élément de `g`. Comme `Array.of_list g.(i)` demande autant d'opérations que le degré du sommet `i`, `Array.map Array.of_list g` demande au total $\sum\limits_{i} \deg(i) = 2m = \mathrm{O}(m)$. Il y a donc bien une complexité $\boxed{\mathrm{O}(n+m)}$ au total.

4. 

5. 

6. 

```
let quadrillage p q =
    let n = p*q in
    let g = Array.make n [] in
    for i = 0 to n - 1 do
        if i mod p <> 0 then g.(i) <- [i - 1];
        if i mod p <> p - 1 then g.(i) <- (i + 1)::g.(i);
        if i >= p then g.(i) <- (i - p)::g.(i);
        if i < n - p then g.(i) <- (i + p)::g.(i)
    done;
    g
```

# II Caractérisation des arbres

7. 
- $s \in C_s$ donc $C_s \neq \emptyset$
- $S_n \subseteq \bigcup C_s$ car si $s \in S_n$ alors $s \in C_s$. $C_s \subseteq S_n$ par définition donc $\bigcup C_s \subseteq S_n$. Donc $S_n = \bigcup C_s$.
- Soient $s, t \in S_n$. Supposons $C_s \cap C_t \neq \emptyset$ et montrons $C_s = C_t$. Comme $C_s \cap C_t \neq \emptyset$, il existe un sommet $u \in C_s \cap C_t$. Comme $u \in C_s$, il existe un chemin $C_1$ de $u$ à $s$. De même, il existe un chemin $C_2$ de $u$ à $t$. En concaténant $C_1$ et $C_2$, on obtient un chemin $C$ de $s$ à $t$. Alors, si $v \in C_s$, la concaténation d'un chemin de $v$ à $s$ et de $C$ donne un chemin de $v$ à $t$, ce qui montre $C_s \subseteq C_t$. De même, on montre $C_t \subseteq C_s$ et donc $\boxed{C_s = C_t}$.

8. Soit $\mathcal{C} = \{\text{ longueur de } C \mid C \text{ est un chemin de } s \text{ à } t\ \}$. Comme $t \in C_s$, $\mathcal{C} \neq \emptyset$. Comme $\mathcal{C}$ est un sous-ensemble non vide de $\mathbb{N}$, il possède un minimum.
Notons $C$ un chemin réalisant ce minimum. Supposons que $C$ passe plusieurs fois par le même sommet $u$. Alors on peut décomposer $C$ en un chemin $C_1$ de $s$ à $u$, puis un chemin $C_2$ partant de $u$ et revenant en $u$, puis un chemin $C_3$ de $u$ vers $t$. Alors, en supprimant $C_2$, obtient un chemin de $s$ vers $t$ (composé de $C_1$ et $C_3$) de longueur strictement inférieure à $C$, ce qui est absurde. Donc les sommets de $C$ sont distincts.

9. Supposons par l'absurde que les extrémités $u, v$ de $a_k$ soient dans la même composante connexe. Alors il existe un chemin $C$ dans $G_k$ de $u$ à $v$. La concaténation de $C$ et de $a_k$ donne un cycle. Ce cycle existe aussi dans $G$, ce qui est absurde pour

un arbre.
Comme il y a initialement $n$ composantes connexes, que chaque ajout d'arête diminue de 1 le nombre de composante connexe et qu'on obtient un arbre $G$ avec 1 composante connexe (car $G$ est un arbre donc connexe donc possède 1 seule composante connexe), $n-1$ arêtes ont été ajoutés : $\boxed{m = n - 1}$.

10.

$(i) \implies (ii)$ Si $G$ un arbre alors $G$ est connexe par définition et $m = n - 1$ par la question précédente.

$(ii) \implies (iii)$ Si $G$ est connexe et $m = n-1$ : supposons que $G$ contienne un cycle $C$. Soit $a$ une arête de $C$. Alors $G - a$ (le graphe obtenu en enlevant $a$ dans $G$) est connexe. En effet : si $u$ et $v$ sont deux sommets de $G$ alors ils sont reliés par un chemin $C_{uv}$ dans $G$ (car $G$ est connexe) et, en remplacant $a$ par $C - a$ dans $C_{uv}$, on obtient un chemin de $u$ à $v$ dans $G - a$.
Ainsi $G - a$ est connexe et possède $n$ sommets et $n - 2$ arêtes, ce qui est absurde d'après Q9.

$(iii) \implies (i)$ Supposons $G$ acyclique et $m = n - 1$. Soient $C_1, ..., C_k$ les composantes connexes de $G$ et $n_1, ..., n_p$ leurs nombres de sommets. D'après Q9, chaque $C_k$ possède $n_k - 1$ arêtes. Donc $G$ possède $\sum_{k=1}^{p}(n_k - 1) = n - p$ arêtes. Comme $m = n - 1$ par hypothèse, $p = 1$. Donc $G$ est connexe et $(i)$ est démontré.

11.

```
let rec representant p s =
    if p.(s) < 0 then s
    else representant p p.(s);;
```

12.

```
let union p s t =
    if p.(s) = p.(t) then p.(t) <- p.(t) - 1;
    if p.(s) < p.(t) then p.(t) <- s
    else p.(s) <- t;;
```

13. Montrons la proposition suivante par récurrence :

$H_k$ : si $\mathcal{P}$ est une partition de $S_n$ construite à partir de $\mathcal{P}_n^{(0)}$ avec au plus $k$ réunions et $X \in \mathcal{P}$ alors $|X| \geq 2^{h(s)}$.

- $H_0$ est vraie car on a alors $h(s) = 0$ et $|X| = 1$ (toutes les parties sont des singletons).
- Soit $k \in \mathbb{N}^*$. Supposons $H_{k-1}$ et considérons $\mathcal{P}$ une partition de $S_n$ construite à partir de $\mathcal{P}_n^{(0)}$ avec $k$ réunions.
La dernière réunion a permis d'obtenir $\mathcal{P}$ à partir d'une partition $\mathcal{P}'$ en réunissant les parties $X$ et $Y$ associées à deux sommets $s$ et $t$, pour obtenir une partie $Z$. On note $h(t)$ la hauteur de $t$ dans $\mathcal{P}$ et $h'(t)$ la hauteur de $t$ dans $\mathcal{P}'$.
Supposons que $t$ ait été choisi comme représentant à l'issue de cette union (le cas où $s$ l'a été est similaire).
D'après $H_{k-1}$, $|Y| \geq 2^{h'(t)}$. Il y a 2 cas : soit $h(t) = h'(t)$ soit $h(t) = h'(t) + 1$.
Si $h(t) = h'(t)$ alors $|Z| \geq |Y| \geq 2^{h'(t)} = 2^{h(t)}$.
Si $h(t) = h'(t) + 1$ alors $h'(s) = h'(t)$ (seule possibilitée pour augmenter la hauteur) et :
$$|Z| = |Y| + |X| \underbrace{\geq}_{H_{k-1}} 2^{h'(t)} + 2^{h'(s)} = 2^{h'(t)+1} = 2^{h(t)}$$
On a donc bien montré $H_k$.

D'après le principe de récurrence, $H_k$ est donc vraie pour tout $k \in \mathbb{N}$.

14. `union p s t` est clairement en $\mathrm{O}(1)$. `representant p s` est en complexité linéaire en la hauteur de l'arbre contenant `s`, qui est $h(s) \leq \log_2(n)$ (d'après Q13), c'est-à-dire $\mathrm{O}(\log(n))$, où $n$ est le nombre de sommets.

15.

```
let est_un_arbre g =
    let r = ref true in
    let n = Array.length g in
    let p = Array.init n (fun i -> -1) in
    for i = 0 to n - 1 do
        let ri = representant p i in
        List.iter (fun j ->
            let rj = representant p j in
            if ri = rj then r := false
            else union p ri rj) g.(i)
    done;
    !r && nombre_aretes g = n - 1
```

# III Algorithme de Wilson

16. Le chemin {`debut = 1; fin = 4; suivant = [|-5; 2; 5; 3; -1; 4|]`} part du sommet 1 pour aller en 2 puis 5 puis 4.

17. Cet algorithme peut ne pas terminer (si on tombe toujours sur un cycle avant de rencontrer $\mathcal{T}$) mais la probabilité que cela arrive est nulle.

18.

```
let marche_aleatoire adj parent s =
    let c = {
        debut = s;
        fin = s;
        suivant = Array.make (Array.length adj) (-1)
    } in
    while parent.(c.fin) = -2 do
        let i = Random.int (Array.length adj.(c.fin)) in
        let v = adj.(c.fin).(i) in
        c.suivant.(c.fin) <- v;
        c.fin <- v;
    done;
    c
```

19.

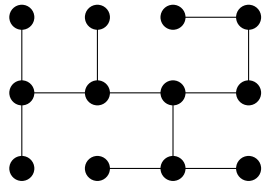
```
let rec greffe parent c =
    let u = ref c.debut in
    while !u <> c.fin do
        let v = c.suivant.(!u) in
        parent.(!u) <- v;
        u := v
    done
```

20.

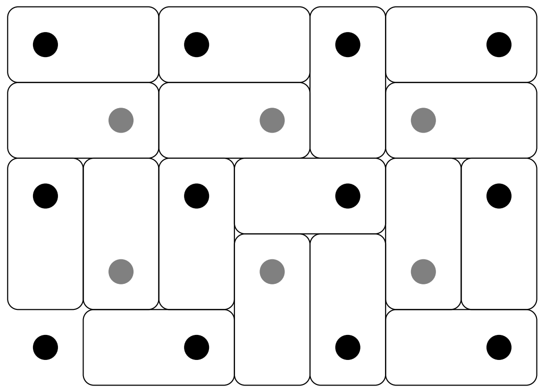
```
let wilson g r =
    let n = Array.length g in
    let adj = adjacence g in
    let parent = Array.make n (-2) in
    parent.(r) <- -1;
    for s = 0 to n - 1 do
        if parent.(s) = -2 then
            let c = marche_aleatoire adj parent s in
            greffe parent c
    done;
    parent
```

# IV Arbres couvrants et pavages par des dominos

21.



22.



23. On peut récupérer les coordonnées $(i, j)$ de $s$ en utilisant la fonction `sommets` et utiliser la direction du domino en $(i, j)$.

24.

```
let coord_noire i = 2*(i mod p), 2*(i/p)
```

25.

```
let sommet_direction s d =
    let i, j = coord_noire s in
    match d with
        | N -> if j = q - 1 then -1 else s + p
        | S -> if j = 0 then -1 else s - p
        | W -> if i = 0 then -1 else s - 1
        | E -> if i = p - 1 then -1 else s + 1
```

26.

```
let phi pavage =
    let n = Array.length pavage in
    let parent = Array.make n (-1) in
    for i = 1 to n - 1 do (* on laisse -1 dans parent.(0) (racine) *)
        let k, l = coord_noire i in
        parent.(i) <- sommet_direction i pavage.(k).(l)
    done;
    parent
```

# Corrigé - X-ENS Informatique A - 2014

## Parti I - Structure d'arbre croissant

### Question 1

Selon la définition d'un arbre croissant, si $t = N(g, x, d)$, on a $x$ qui est plus petit que tous les éléments de $g$ et de $d$. Ainsi le minimum est la racine :

```
let minimum (N (_,m,_)) = m;;
```

### Question 2

On parcoure l'arbre de manière récursive en vérifiant à chauqe noeud $N(g, x, d)$ que $x$ minore $g$ et $d$. Pour cela, il suffit de vérifier que $x$ est plus petit que les racines de ces arbres s'ils sont non vides, ce qui s'effectue en temps constant. La complexité correspond donc au nombre d'appels récursifs, i.e. au nombre de noeuds de $t$ : $O(|t|)$.

```
let rec est_un_arbre_croissant t =
    let minore x t =
        match t with
        | E -> true
        | N (_,y,_) -> x <= y
    in
    match t with
    | E -> true
    | N (g,x,d) -> minore x g && minore x d
        && est_un_arbre_croissant g
        && est_un_arbre_croissant d;;
```

### Question 3

Soit $H_n$ : il existe $n!$ arbres croissants à $n$ noeuds étiquetés par $n$ entiers distincts.
Montrons $H_n$ par récurrence forte sur n. $H_1$ est clairement vérifiée.
Soit $n \in \mathbb{N}^*$. Supposons $H_k$ pour tout $k \in \{1, ..., n\}$.
Soit $E$ un ensemble de $n + 1$ entiers distincts.
Pour obtenir un arbre croissant $t$ à $n + 1$ sommets étiquetés par des entiers de $E$, il faut choisir sa racine et ses deux sous-arbres. Il n'y a qu'un seul choix pour la racine de $t$ : le minimum $m$ de $E$.
Soit $k \in \{0, 1, ..., n\}$. Comptons le nombre $a_k$ de façon de choisir $t$ avec un sous-arbre à $k$ sommets :

1. Il faut choisir $k$ sommets parmi $E - \{m\}$ : $\binom{n}{k}$ possibilités.
2. Il faut choisir un sous-arbre gauche croissant avec ces $k$ sommets : $k!$ possibilités d'après $H_k$.
3. Il faut choisir un sous-arbre droit croissant avec les $n-k$ sommets restants : $(n - k)!$.

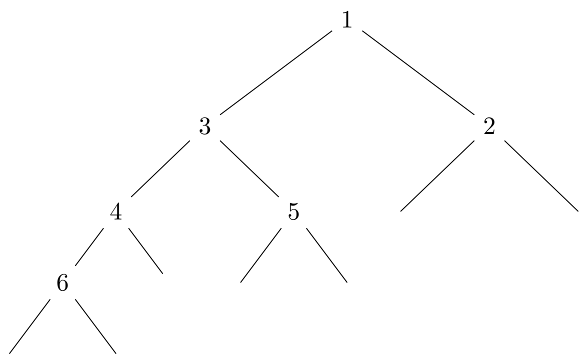
Le nombre de choix pour $t$ est donc :

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{n} \binom{n}{k} k!(n-k)! = \sum_{k=0}^{n} n! = \boxed{(n+1)!}$$

On a bien démontré $H_{n+1}$. D'après le principe de récurrence, $H_n$ est vraie pour tout $n \in \mathbb{N}^*$.

## Partie II - Opérations sur les arbres croissants

### Question 4




### Question 5

Les noeuds de $t$ sont de deux sortes :
- Soit il s'agit de noeuds de $t_1$ ou de $t_2$ reproduits tels quels.
- Soit il sont construit lors d'un appel récursif. Dans ce cas si c'est le noeud $N(g_1, x_1, d_1)$ qui est déconstruit alors l'appel récursif porte sur $d_1$ et $t_2$, qui contiennent un noeud $x_1$ de moins que $t_1$ et $t_2$, et on construit un noeud d'étiquette $x_1$.
  Ainsi, le nombre d'occurrences de $x_1$ est stable.
  Le cas de l'autre appel récursif est symétrique.

Les occurrences sont donc en bijection entre celles de $t$ et celles de $(t_1, t_2)$.

### Question 6

On fusionne l'arbre $t$ avec un arbre qui ne contient qu'un noeud d'étiquette $x$, ainsi, après fusion, on obtiendra, selon la question précédente, un arbre $t'$ tel que $occ(x, t') = occ(x, t) + occ(x, N(E, x, E)) = 1 + occ(x, t)$.

```
let ajoute x t = fusion t (N (E,x,E));;
```

### Question 7

On sait déjà que si $t = N(g, x, d)$ alors $x$ est une occurrence du minimum de $t$. On a donc $occ(x, g) + occ(x, d) = occ(x, t) - 1$ et on peut fusionner $g$ et $d$ pour obtenir l'arbre $t'$ ayant une occurrence de moins du minimum.

```
let supprime_minimum (N(g,_,d)) = fusion g d;;
```

### Question 8

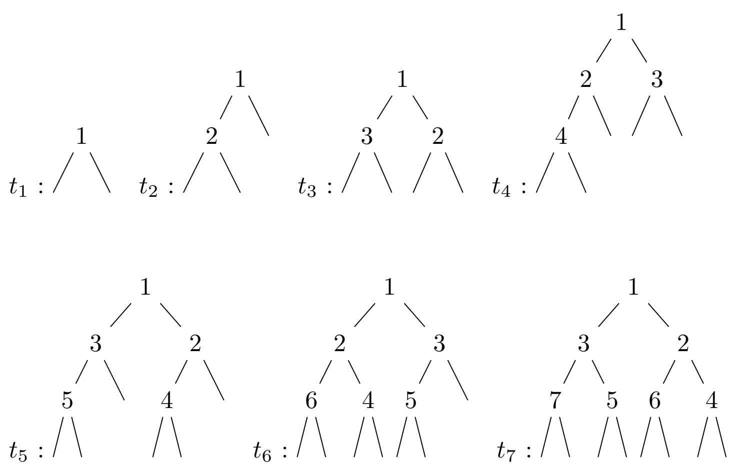
On utilise directement la fonction `ajoute` de la question 6 qui a été formulée pour être compatible avec la construction demandée (fusion à droite) :

```
let ajouts_successifs v =
    let a = ref E in
    for i = 0 to vect_length v - 1 do
        a := ajoute v.(i) !a
    done;
    !a;;
```

## Question 9

Considérons $t = N(N(\cdots N(E, p+k, E)\cdots, p+2,), p+1, E)$, c'est-à-dire un arbre peigne qui ne compte qu'un seul chemin de la racine $p{+}1$ au nœud $p{+}k$. On ajoute $p$ à cet arbre. Comme il est plus petit que tous les nœud, il va forcément être placé comme nouvelle racine de la fusion et $t$ va être le sous arbre gauche de la fusion. Ainsi $t'$ issu de la fusion a la forme suivante $t' = N(t, p, E)$ et est aussi un peigne mais augmenté d'un cran.
On en déduit que l'arbre issu de l'ajout de $x_0 = n > x_1 = n{-}1 > \cdots > x_{n-1} = 1$ est un peigne, et donc qu'il est de hauteur $n \geq n/2$.

## Question 10




Un peu d'expérimentation suggère que les arbres $t_n$ vérifient la propriété (*) : pour tout nœud de taille $p \geq 1$ le fils gauche est de taille $\left\lceil \frac{p-1}{2} \right\rceil = \left\lfloor \frac{p}{2} \right\rfloor$ et le fils gauche est de taille $\left\lfloor \frac{p-1}{2} \right\rfloor$
Lors d'un ajout les différentes adjonctions se font en ajoutant un élément supérieur à ceux de l'arbre, on est donc dans le cas de la troisième (ou deuxième) ligne de la définition de `fusion`.
On procède par récurrence.

1. Lors de l'ajout à un nœud de taille 0 on obtient `N(E,n,E)`, qui vérifie (*).
2. Lors de l'ajout nœud de taille $p \geq 1$ qui vérifie $(*)$
   - la taille devient $p+1$
   - le fils droit est l'ancien fils gauche,
     il vérifie (*) et est de taille $\left\lfloor \frac{p}{2} \right\rfloor = \left\lfloor \frac{(p+1)-1}{2} \right\rfloor$

- le fils gauche est l'ancien fils droit auquel on ajoute un élément plus grand : par récurrence il vérifie (*) et est de taille $\lfloor \frac{p-1}{2} \rfloor + 1 = \lfloor \frac{p+1}{2} \rfloor$
- Ainsi le nouveau nœud vérifie (*)

On prouve alors, toujours par récurrence, que si $2^p \leq n < 2^{p+1} - 1$ alors la hauteur de $t_n$ est $p + 1$

# Partie III - Analyse

## Question 11

Pour garantir la complexité en $O(|t|)$ il faut effectuer en même temps le calcul de taille et le calcul de potentiel.

```
let potentiel t =
    let rec aux t =
        match t with
        | E -> 0, 0
        | N(g,_,d) ->
            let taille_g, pot_g = aux g in
            let taille_d, pot_d = aux d in
            1+taille_g+taille_d,
                (if taille_g < taille_d then 1 else 0)+pot_g+pot_d
    in snd (aux t);;
```

## Question 12

On effectue une récurrence sur la taille de la fusion
- On a $\Phi(E) = 0$ et $\log |E| = 0$ et la fusion de $t$ et de $E$ est $t$.
  On en déduit $0 = C(t, E) \leq \Phi(t) + \Phi(E) - \Phi(t) + 2(\log |t| + \log |E|) = 2 \log |t|$.
  La formule (1) est symétrique ; ainsi elle est vérifiée si $t_1$ ou $t_2$ est vide.
- On suppose maintenant $t_1$ et $t_2$ non vides.
  Soit $t$ le résultat de la fusion de $t_1 = N(g_1, x_1, d_1)$ et de $t_2 = N(g_2, x_2, d_2)$.
  On suppose $x_1 \leq x_2$.
  Alors $t = N(t_2', x_1, g_1)$ où $t_2'$ est le résultat de la fusion de $d_1$ et de $t_2$.
  On a $\Phi(t) = \Phi(t_2') + \Phi(g_1) + \alpha$ avec $\alpha = 1$ si $t$ est lourd et $\alpha = 0$ sinon.
  De même $\Phi(t_1) = \Phi(d_1) + \Phi(g_1) + \alpha_1$ avec $\alpha_1 = 1$ si $t_1$ est lourd ou $\alpha_1 = 0$n.
  On a toujours $\log |t_1| \geq \log |d_1|$.
  Si $\alpha_1 = 0$ alors $|g_1| \geq |d_1|$ d'où $|t_1| = |g_1| + |d_1| + 1 \geq 2|d_1|$ donc
  $\log |t_1| \geq \log |d_1| + 1$. Dans tous les cas on a $\log |d_1| + 1 - \alpha_1 \leq \log |t_1|$.
  L'hypothèse de récurrence donne
  $C(d_1, t_2) \leq \Phi(d_1) + \Phi(t_2) - \Phi(t_2') + 2(\log |d_1| + \log |t_2|)$ d'où

$$\begin{aligned}
C(t_1,t_2) &= C(d_1,t_2)+1\\
&\le \Phi(d_1)+\Phi(t_2)-\Phi(t_2')+2(\log|d_1|+\log|t_2|)+1\\
&\le \Phi(t_1)-\Phi(g_1)-\alpha_1+\Phi(t_2)-\Phi(t_2')+2(\log|d_1|+\log|t_2|)+1\\
&\le \Phi(t_1)+\Phi(t_2)-\Phi(g_1)-\Phi(t_2')+2(\log|d_1|+\log|t_2|)+1-\alpha_1\\
&\le \Phi(t_1)+\Phi(t_2)-\Phi(t)-\alpha+2(\log|d_1|+\log|t_2|)+1-\alpha_1\\
&\le \Phi(t_1)+\Phi(t_2)-\Phi(t)+\log|d_1|+1-\alpha_1+\log|d_1|+2\log|t_2|\\
&\le \Phi(t_1)+\Phi(t_2)-\Phi(t)+\log|t_1|+\log|d_1|+2\log|t_2|\\
&\le \Phi(t_1)+\Phi(t_2)-\Phi(t)+2(\log|t_1|+\log|t_2|)
\end{aligned}$$

(1) est vérifiée.
Le cas $x_1 > x_2$ se fait de même.

## Question 13

Soit $C$ le cout total, on a $C = \sum_{i=1}^{n} C(t_{i-1}, N(E,x_i,E)) \le \sum_{i=1} \Phi(t_{i-1}) + \Phi(N(E,x_i,E)) - \Phi(t_i) + 2(\log|t_{i-1}| + \log|N(E,x_i,E)|) \le \Phi(t_0) - \Phi(t_n) + 2\sum_{i=1}^{n} \log|t_{i-1}| = 2\sum_{i-1}^{n} \log i - \Phi(t_n) \le 2n\log n$.
Donc $C = O(n\log n)$.

## Question 14

Lors d'un ajout d'un élément, on fusionne $t$ avec un arbre ne contenant aucun nœud mis à part la racine. Le cout de la fusion correspond donc exactement au nombre de nœuds sur le chemin le plus à droite de $t$ plus 1 (le noeud ajouté).
Supposons $n = 2p-1$, on pose $x_0 = p, x_1 = p+1, x_2 = p-1, x_3 = p+2, x_4 = p-2, \cdots, x_{n-4} = 2p-1, x_{n-3} = 1, x_{n-2} = 2p, x_{n-1} = n$.
L'arbre $t_{n-1}$ obtenu a un chemin tout à droite qui contient les nœuds $1, 2, \cdots, p$ et ces nœuds ont pour fils gauche des nœuds simples contenant, dans l'ordre, les nœuds $2p, \cdots, p+1$.
Le cout de fusion avec $N(E,n,E)$ est donc de $p+1 \ge n/2$.
Le cout total de la construction étant en $O(nlogn)$ c'est que ce surplus de cout sur une fusion est compensée par des fusions moins couteuses. Pour la construction globale tout se passe comme si les fusions avait un cout en $O(logn)$, on a *amorti* le surcout.

## Question 15

Soit $C$ le cout total, on a $C = \sum_{i=0}^{n-1} C(g_i,d_i) \le \sum_{i=1}^{n-1} \Phi(g_i) + \Phi(d_i) - \Phi(t_{i+1}) + 2(\log|g_i| + \log|d_i|) \le \sum_{i=0}^{n-1} \Phi(t_i) - \Phi(t_{i+1}) + 4\log|t_i| = \Phi(t_0) - \Phi(t_n) + 4\sum_{i=0}^{n-1} \log|t_i| \le 2n + 4n\log n$.
Donc $C = O(n\log n)$.

# Partie IV - Applications

## Question 16

On remarque que dans la construction de la question 15 on passe de $t_i$ à $t_{i+1}$ en supprimant le minimum. On peut donc effectuer cette construction et à chaque étape stocker dans le tableau le minimum courant. On aura ainsi trié le tableau pour un cout en $O(n \log n)$. De plus, on note que chaque arbre aura une taille plus petite que $2n+1$ et qu'on ne peut garder en mémère que deux arbres : l'arbre courant et le prochain. Ainsi, en espace la complexité est en $O(n)$ (*non demandé mais cela semble crucial dans ce genre d'algorithmes qui allouent des structures*).

```
let tri v =
    let t = ref (ajouts_successifs v) in
    let n = vect_length v in
    for i = 0 to n -1 do
        v.(i) <- minimum !t;
        t := supprime_minimum !t
    done;;
```

## Question 17

On remarque facilement que $t_i^j$ contient $2^i$ nœuds et donc $\log |t_i^j| \leq i+2$ car $|t_i^j| = 2^{i+1}+1 \leq 2^{i+2}$.
Soit $C$ le cout total on a $C = \sum_{i=0}^{k-1} \sum_{j=0}^{2^{k-i-1}} C(t_i^{2j}, t_i^{2j+1}) \leq \sum_{i=0}^{k-1} \sum_{j=0}^{2^{k-i-1}-1} \Phi(t_i^{2j}) + \Phi(t_i^{2j+1}) - \Phi(t_{i+1}^j) + 2(\log |t_i^{2j}| + \log |t_i^{2j+1}|) \leq 4 \sum_{i=0}^{k-1} 2^{k-i-1}(i+2) - \Phi(t_k^0) \leq 2^k \times 2 \sum_{i=0}^{k-1} \frac{i+2}{2^i}$
Or $\sum_{i=0}^{k-1} \frac{i+2}{2^i} \xrightarrow{k \to +\infty} 2$ donc est bornée à partir d'un certain rang. Ainsi $C = O(2^k)$.

# Question 18

On effectue directement le calcul précédent à l'aide d'une fonction auxillaire portant sur $i$ et sur $j$. Aucun appel récursif n'est effectué inutilement deux fois car on ne fusionne jamais deux fois un arbre dans la construction précédente.

```
let construire v =
    let rec aux i j =
        if i = 0
        then N(E,v.(j),E)
        else fusion (aux (i-1) (2*j)) (aux (i-1) (2*j+1))
    in
    let n = vect_length v in
    let k = log2 n in
    aux k 0;;
```