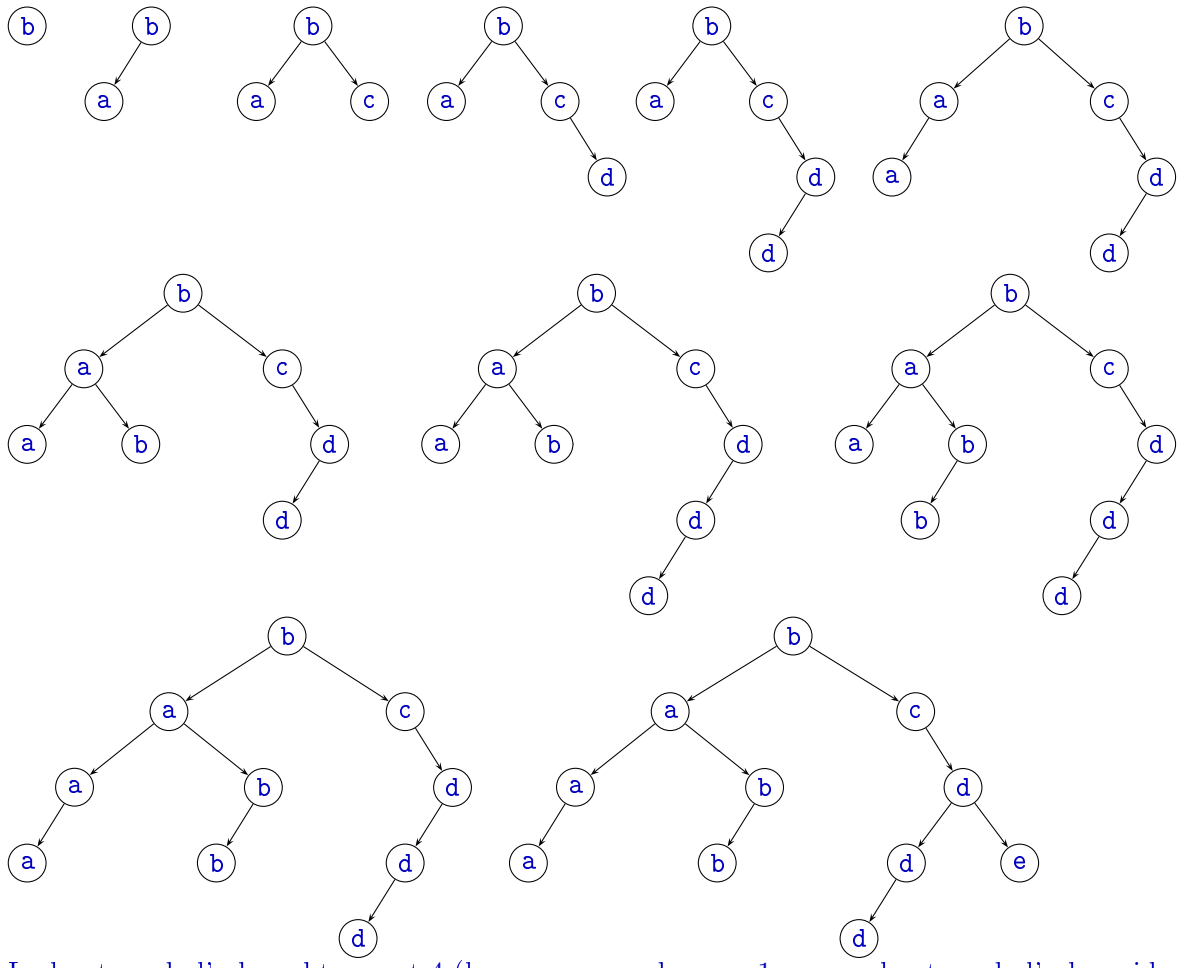
❑ **Exercice 1** : *Doublons autorisés dans un arbre binaire de recherche* 🎓 Oraux CCINP

Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche et pour le cas d'égalité on choisira le sous-arbre gauche. On ne cherchera pas à équilibrer les arbres.

1. Rappeler la définition d'un arbre binaire de recherche.

   Un arbre binaire de recherche sur un ensemble d'étiquettes E (totalement ordonné), peut se définir inductivement par l'axiome $\varnothing$ (arbre vide), et la règle d'inférence d'arité 2 : $(g, d) \mapsto (g, x, d)$ où $x \in E$ et toute étiquette de g est inférieure à x et toute étiquette de d est supérieure à x.

2. Insérer successivement et une à une dans un arbre binaire de recherche initialement vide toutes les lettres du mot `bacddabdbae`, en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?



   La hauteur de l'arbre obtenu est 4 (la programme donne -1 comme hauteur de l'arbre vide.)

3. Montrer que le parcours en profondeur infixe d'un arbre binaire de recherche de lettres est un mot dont les lettres sont rangées dans l'ordre croissant. On pourra procéder par induction structurelle.

On procède par induction structurelle, pour tout ABR a, on note $P(a)$ la propriété « le parcours infixe de a est un mot dont les lettres sont rangées dans l'ordre croissant ».

— $P(\varnothing)$ est vraie et donc la propriété P est vérifiée pour tous les axiomes.

— Montrons à présent la conservation de la propriété P par application de la règle d'inférence, si g et d sont deux ABR vérifiant P, et x une lettre telle que toutes les lettres de g sont avant x et toutes les lettres de d sont après x dans l'ordre alphabétique. Le parcours infixe de l'ABR (g, x, d) est par définition le parcours infixe de g suivi de x suivi du parcours infixe de d. Par hypothèse d'induction, les lettres du parcours de g et celles du parcours de d sont rangées par ordre croissant. Comme de plus x est après toute les lettres de g et avant toutes celles de d, le parcours de (g, x, d) est bien formé de lettres rangées par ordre croissant.

Par application du principe d'induction structurelle, pour tout ABR a, $P(a)$ est vraie.

4. Proposer un algorithme qui permet de compter le nombre d'occurrences d'une lettre dans un arbre binaire de recherche de lettres. Quelle est sa complexité ?

On part de la racine, à chaque étape, on ajoute 1 au nombre d'occurence si l'étiquette est la lettre cherchée et on descend dans le sous arbre gauche si l'étiquette est inférieure ou égale à la racine et dans le sous arbre droit sinon. On s'arrête en rencontrant une feuille. Comme à chaque étape on descend d'un niveau dans l'orbre la complexité est en $O(h)$ où h est la hauteur de l'arbre.

5. Pour l'implémentation on propose d'utiliser le type `abr` suivant :

```
type abr = Vide | Noeud of abr*char*abr;;
```

Ecrire la fonction `insere abr -> char -> abr` qui prend en argument un arbre `abr` et une lettre et renvoie l'arbre obtenu en insérant cette lettre dans `abr`.

```
let rec insere a l =
  match a with
  | Vide -> Noeud(Vide,l,Vide)
  | Noeud(g,x,d) -> if l<=x then Noeud(insere g l, x, d) else Noeud(g, x,
  ↪  insere d l);;
```

6. Proposer une implémentation en OCaml pour l'algorithme de la question 4

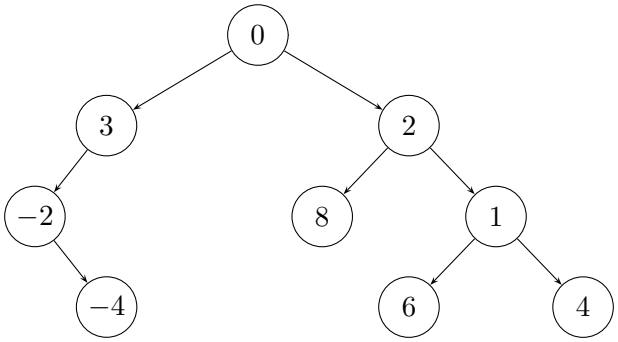
```
let rec compte arbre lettre =
  match arbre with
  | Vide -> 0
  | Noeud(g,x,d) -> if lettre<x then compte g lettre else
                    if lettre=x then 1+ compte g lettre  else
                        compte d lettre;;
```

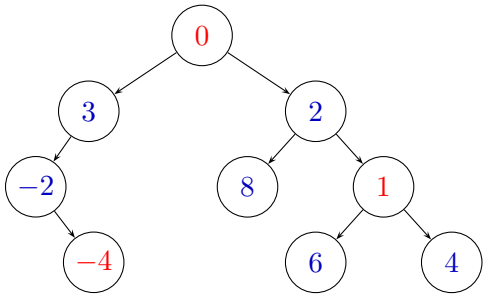
❑ Exercice 2 : *Minima locaux dans des arbres*

🎓 Oraux CCINP

Dans cet exercice, on considère des arbres étiquetés par des entiers relatifs deux à deux distincts. On dit qu'un noeud est un *minimum local* d'un arbre si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels fils.

1. Déterminer le ou les minima locaux de l'arbre A suivant :





2. Donner la définition inductive des arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre A ?

Un arbre binaire sur un ensemble d'étiquettes E, peut se définir inductivement par l'axiome $\varnothing$ (arbre vide), et la règle d'inférence d'arité 2 : $(g,d) \mapsto (g,x,d)$ où $x \in E$.
La hauteur d'un arbre binaire se définit par $h(\varnothing) = -1$ et $h(g,e,d) = 1 + \max\{h(g), h(d)\}$.
L'arbre A a une hauteur de 3.

3. Montrer que tout arbre non vide possède un minimum local.

On effectue une démonstration par récurrence forte sur la taille notée n de l'arbre, pour tout $n \in \mathbb{N}^*$, soit $P(n)$ la propriété « un arbre binaire de taille n possède un minimum local ».

— Si $n = 1$ alors la racine est un minimum local car elle ne possède ni fils ni père.

— Soit $n \in \mathbb{N}^*$ tel que $P(n)$ soit vraie, on considère un arbre A de taille $n+1$ et de racine d'étiquette r, alors soit la racine est un minimum local et $P(n+1)$ est vérifiée. Soit la racine possède un fils qui est lui est strictement inférieure, on suppose sans perdre de généralité qu'il s'agit du sous arbre gauche et on note l'étiquette de sa racine g. Ce sous arbre gauche est de taille strictement inférieure à n et il possède donc un minimum local. Si ce minimum local est la racine alors il s'agit aussi d'un minimum local de l'arbre A puisque $r > g$. Dans tous les cas $P(n+1)$ est vérifiée.

Donc, tout arbre binaire non vide possède un minimum local.

4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.

On propose un algorithme basé sur le raisonnement de la question précédente :

— si l'arbre est réduit a une feuille alors on termine en renvoyant l'étiquette

— sinon, on compare l'etiquette de la racine avec l'étiquette de la racine du sous arbre gauche, notée g, et l'étiquette de la racine du sous arbre droit notée d (lorsque ces sous arbres existent). Si $r < g$ et $r < d$ alors r est un minimum local et l'algorithme termine. Sinon on effectue un appel récursif sur le sous arbre gauche s'il existe et si $r > g$ et sur le sous arbre droit sinon.

5. Ecrire une implémentation de cet algorithme en OCaml en utilisant le type arbre binaire suivant :

```
type ab =
 |Vide
 |Noeud of ab *  int * ab;;
```

```
let rec trouve_min arbre =
  match arbre with
  | Vide -> failwith "Pas de minimum local dans un arbre vide !"
  | Noeud(Vide,e,Vide) -> e
  | Noeud(Noeud(gg,eg,gd),e,Vide) -> if e<eg then e else trouve_min
  ↪  (Noeud(gg,eg,gd))
  | Noeud(Vide,e,Noeud(gd,ed,dd)) -> if e<ed then e else trouve_min
  ↪  (Noeud(gd,ed,dd))
  | Noeud(Noeud(gg,eg,gd),e,Noeud(dg,ed,dd)) -> if (e<eg && e<ed) then e
  ↪  else if e>eg then trouve_min (Noeud(gg,eg,gd)) else trouve_min
  ↪  (Noeud(dg,ed,dd))
```

6. Ecrire une fonction `min_locaux ab -> int list` qui prend en argument un arbre binaire et renvoie la liste de tous ses minima locaux.

```
let rec min_locaux arbre =
  let rec aux_ml arbre parent =
  match arbre with
  | Vide -> []
  | Noeud(Vide,e,Vide) -> if (e<parent) then [e] else []
  | Noeud(Noeud(gg,eg,gd),e,Vide) -> if (e<parent && e<eg) then e::(aux_ml
  ↪  (Noeud(gg,eg,gd)) e) else (aux_ml (Noeud(gg,eg,gd)) e)
  | Noeud(Vide,e,Noeud(dg,ed,dd)) -> if (e<parent && e<ed) then e::(aux_ml
  ↪  (Noeud(dg,ed,dd)) e) else (aux_ml (Noeud(dg,ed,dd)) e)
  | Noeud(Noeud(gg,eg,gd),e,Noeud(dg,ed,dd)) -> if (e<parent && e<eg &&
  ↪  e<ed) then e::(aux_ml (Noeud(gg,eg,gd)) e)@(aux_ml (Noeud(dg,ed,dd))
  ↪  e) else (aux_ml (Noeud(gg,eg,gd)) e)@(aux_ml (Noeud(dg,ed,dd)) e)
  in aux_ml arbre Int.max_int
```

❑ Exercice 3 : *Tableaux autoréférents*

Oraux CCINP

On dit qu'un tableau `tab` de taille n est *autoréférent* si pour tout entier `i` $\in [\![0; n-1]\!]$, `tab.(i)` est le nombre d'occurences de `i` dans `tab`. Par exemple, le tableau `ex=[| 1; 2; 1; 0 |]` est autoréférent, en effet :

- `ex.(0) = 1` et `0` apparaît bien une fois dans le tableau
- `ex.(1) = 2` et `1` apparaît bien deux fois dans le tableau
- `ex.(2) = 1` et `3` apparaît bien une fois dans le tableau
- `ex.(3) = 0` et `3` n'apparaît pas dans le tableau

1. Justifier rapidement que si `tab` est un tableau autoréférent de taille `n` alors les éléments de `tab` sont tous inférieurs ou égaux à `n`

   Une valeur apparraît au plus n fois dans un tableau de taille n.

2. Montrer que pour $n \in [\![1; 3]\!]$, il n'existe aucun tableau auto référent de taille n.

   Il suffit de tester les possibilités, en utilisant la question précédente.

3. Déterminer un autre tableau autoréférent de taille 4 que celui donné en exemple.

   En testant les possiblités dans le cas $n = 4$, on trouve `[| 2; 0; 2; 0 |]`

4. Soit `n`≥ 7, on définit le tableau `tab` de taille `n` par :
   - `tab.(0) = n-4`
   - `tab.(1) = 2`
   - `tab.(2) = 1`
   - `tab.(n-4) = 1`
   - `tab.(i) = 0` si `i` $\notin \{0, 1, 2, n-4\}$

   Prouver que `tab` est autoréférent

   On vérifie :
   - `tab.(0) = n-4` et toutes les cases du tableau valent `0` sauf 4 cases, celles d'indices 0 (car $n - 4 \neq 0$), 1, 2 et $n - 4$.
   - `tab.(1) = 2` et `1` apparaît bien deux fois dans le tableau aux indices 2 et $n - 4$. (car $n - 4 \neq 1$)
   - `tab.(2) = 1` et `2` apparaît bien une fois dans le tableau (à l'indice 1)
   - `tab.(n-4) = 1` et `n-4` apparaît bien une seule fois dans le tableau par comme $n \leqslant 7$, $n - 4 \leqslant 3$.
   - `tab.(i) = 0` si `i` $\notin \{0, 1, 2, n-4\}$ et aucune de ces valeurs n'apparaît dans le tableau

   Donc ce tableau est bien autoréférent.

5. Montrer que si `tab` est un tableau autoréférent de taille `n` alors la somme des éléments de `tab` vaut `n`. La réciproque est-elle vraie ?

   Soit t un tableau autoréférent de taille n,
   $\sum_{k=0}^{n-1} t_k = \sum_{k=0}^{n-1} \text{occ}(k)$, où $\text{occ}(k)$ est le nombre d'occurences de k dans t.
   Et comme les seules valeurs présentes dans t sont les entiers appartenant à $[\![0; n-1]\!]$ cette somme vaut n.

6. Ecrire en OCaml une fonction `est_autoreferent int array -> bool` qui prend en argument un tableau d'entiers et renvoie `true` si ce tableau est autoréférent et `false` sinon. On attend une complexité en $O(n)$ où n est la taille du tableau.

```
let est_autoreferent tab =
  let n = Array.length tab in
  let occ = Array.make n 0 in
  try
  for i=0 to n-1 do
    if tab.(i)<0 || tab.(i)>n-1 then raise Exit else
    occ.(tab.(i)) <- occ.(tab.(i)) + 1
  done;
  for i=0 to n-1 do
    if tab.(i) <> occ.(i) then raise Exit;
  done;
  true
  with Exit -> false;
;;
```

On cherche maintenant à construire un tableau autoréférent de taille `n` en utilisant un algorithme de recherche par retour sur trace (*backtracking*) qui valide une solution partielle construite jusqu'à un index `idx` donné, on propose pour cela la fonction suivante :

```
    if tab.(i)=elt then res := !res + 1;
  done;
  !res;;

  let partiel t  k =
    let n = Array.length t in
    let cpt = Array.make  n  0 in
    try
    for i=0 to k-1 do
      cpt.(t.(i)) <- cpt.(t.(i)) + 1
    done;
    for i=0 to k-1 do
      if cpt.(i) > t.(i) then  raise Exit;
      done;
```

7. Ecrire une fonction de validation partielle qui pour le moment renvoie toujours `true` et tester cette fonction pour de petites valeurs de `n` (attention, il faut aussi écrire la fonction d'affichage du tableau), que constater vous ?

   ```
   let valide_partielle tab idx = true;;
   ```
   Ne fonctionne que sur de petites valeurs de n ($n \leq 9$) faute d'une rapidité suffisante.

8. Proposer et implémenter des améliorations de la fonction de validation partielle en utilisant les résultats établis sur les tableaux autoréférents aux questions précédentes.

On peut proposer diverses améliorations :

— Vérifier que la somme partielle jusqu'à l'indice `idx` ne dépasse pas la somme du tableau

— Vérifier après avoir affecté une case qu'il n'y a déjà pas plus d'occurrence de la valeur dans le tableau

— Vérifier qu'en ajoutant à la somme courante le nombres de cases restant à remplir et différentes de 0 on ne dépasse pas la taille du tableau

```
  let partiel t  k =
    let n = Array.length t in
    let cpt = Array.make  n  0 in
    try
    for i=0 to k-1 do
      cpt.(t.(i)) <- cpt.(t.(i)) + 1
    done;
    for i=0 to k-1 do
      if cpt.(i) > t.(i) then  raise Exit;
      done;
    true;
    with  Exit -> false;;

let valide_partielle tab idx =
  let n = Array.length tab in
  (* Vérifier que la somme + cases non=0 restant à remplir ne dépasse pas
  ↪  la taille du tableau *)
  if ((somme tab idx + (n-1-idx -(tab.(0) - compte tab 0 idx)) > n) || (not
  ↪  (partiel tab idx))) then false else true;;
```

On obtient une réponse quasi immédiate même pour $n \simeq 100$ (et on constate que le tableau autoréférent de la question 4 est la seule solution.)

❑ **Exercice 4** : *Convergence d'une suite*

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = e - 1 \\ u_{n+1} = (n+1)\,u_n - 1 \end{cases}$$

On note

$$S_n = \sum_{k=0}^{n} \frac{1}{k!}$$

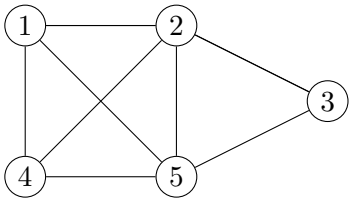
On pourra utiliser sans justification le résultat suivant : pour tout $n \in \mathbb{N}$: $S_n \leq e \leq S_n + \dfrac{1}{n\,n!}$

1. Montrer que $e = \lim\limits_{n\to+\infty} S_n$
2. Monter que pour tout $n \in \mathbb{N}$, $u_n = n!(e - S_n)$
3. En déduire que $(u_n)_{n\in\mathbb{N}}$ converge et donner sa limite.
4. Ecrire en langage C, une fonction `main` qui prend un entier n en argument sur la ligne de commande et affiche les n premières valeurs de la suite u_n. On utilisera le type `double` pour les flottants et on pourra utiliser la valeur `M_E` de `<math.h>` pour représenter le nombre e.
5. Tester votre fonction pour $n = 17$, le comportement observé est-il conforme à celui établi à la question 3 ?
6. Tester votre fonction pour $n = 25$, commenter le résultat obtenu.

❑ **Exercice 5** : *Triangle dans un graphe*

On considère un graphe non orienté $G = (S, A)$ où $A = \{1, \ldots, n\}$. On dit qu'un sous ensemble de V à trois éléments $\{x, y, z\}$ est un *triangle* de G lorsque $\{x, y\}, \{y, z\}$ et $\{x, z\}$ appartiennent à A. Par exemple, dans

le graphe g suivant, $\{1, 2, 4\}$ est un triangle.



1. Donner tous les autres triangles du graphe g.
2. Rappeler la défintion d'un graphe complet. Donner le nombre de triangle d'un graphe complet à n sommets.
3. On dit qu'un graphe est bipartite lorsqu'on il existe une partition de l'ensemble des sommets S en deux ensembles S_1 et S_2 tel que tout arête ait un élément dans S_1 et l'autre dans S_2. Dessiner un graphe bipartite à 6 sommets et 9 arêtes.
4. Montrer qu'un graphe bipartite ne contient pas de triangles.
5. Afin de lister les triangles d'un graphe, on propose de tester si chaque partie de A à trois éléments est un triangle. Indiquer la complexité d'un tel algorithme.
6. Dans cette question uniquement, on utilise des graphes représentés par matrice d'adjacence à l'aide du type structuré en C :

```c
// Nombre maximal de sommets
#define NMAX 100
// La matrice d'adjacence du graphe
typedef bool graphe[NMAX][NMAX];
```

   la fonction d'ajout d'un arc dans un tel graphe s'écrit :

```c
void cree_arete(graphe g, int i, int j){
    g[i][j] = true;
    g[j][i] = true;}
```

   Ecrire la fonction `liste_triangle`, de prototype `void liste_triangle(graphe g, int n)` qui prend en argument un graphe `g` et son nombre de noeud `g` et affiche sur la sortie standard les triangles de ce graphe en utilisant l'algorithme décrit à la question précédente.
7. Un autre algorithme possible pour lister les triangles d'un graphe consiste pour chaque arête $\{x, y\}$ à chercher l'intersection de l'ensemble des sommets z adjacent à x et à y. Donner la complexité de cet algorithme si on suppose que l'intersection est calculée en temps linéaire du nombre de sommets.
8. Ecrire en OCaml, une fonction `intersection int list -> int list -> int list` qui calcule en temps linéaire l'intersection de deux listes *en les supposant triées*.
9. Dans la suite de l'exercice, on utilise des graphes représentés par liste d'adjacence en OCaml avec le type :

```ocaml
type graphe = {
  taille : int;
  ladj : int list array};;
```

   Et on donne la fonction permettant de créer un graphe de taille donnée `n` :

```ocaml
let cree_graphe n =
  {taille=n; ladj = Array.make n []};;
```

   Ecrire une fonction permettant d'ajouter une arête dans un graphe en *maintenant triée* les listes d'adjacence.
10. Implémenter l'algorithme décrit à la question **??** pour lister les triangles d'un graphe.