Attention

- Ce fichier contient des exercices de colles d'informatique pour les MP2I.
- Certains de ces exercices n'ont jamais été donné et n'ont donc jamais été "testé".
- Les programmes mystères en C et en OCAML ne vérifient pas les pratiques de bonne programmation (puisqu'on cherchait à obfusquer leur comportement).
- Ce n'est pas rangé.
- Ce n'est pas corrigé, mais si on me demande une correction préciser j'essaierai d'y répondre.

Exercice 1 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$. On note V le booléen vrai et F le booléen faux.
On considère l'algorithme ci-dessous, paramétré par une valeur entière mystère m.

Algorithme 1 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$
Hypothèse : ??
Sortie : ??

```
1  Soit W un tableau de taille n dont chaque case est initialisée F ;
2  W[n − 1] ← V ;
3  pour i = 0 à m faire
4      j ← 0 ;
5      tant que W[j] = F faire
6          j ← j + 1 ;
7      si T[j − 1] < T[j] alors
8          Échanger le contenu de T[j] et T[j − 1] ;
9      W[j − 1] ← V ;
10 retourner T[0] ;
```

Q. 1 Donner les hypothèses éventuelles et les spécifications de l'algorithme 1.

Q. 2 Proposer un algorithme le plus proche possible de l'algorithme 1 (les opérations d'échange doivent avoir lieu dans le même ordre) mais n'utilisant qu'une boucle **pour**.

Exercice 2 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$. On considère l'algorithme ci-dessous, paramétré par une valeur entière mystère m. On désigne par $a//b$ le quotient de la division euclidienne de a par b.

Algorithme 2 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$
Hypothèse : ??
Sortie : ??

```
1 Soit C un tableau de taille n dont chaque case est initialisée 0 ;
2 pour i = 0 à n − 1 faire
3     pour j = 0 à n − 1 faire
4         si T[i] < T[j] alors
5             C[j] ← C[j] + 1 ;
6 k ← 0 ;
7 while C[k] ≠ n//2 do
8     k ← k + 1 ;
9 retourner T[k] ;
```

Q. 1 Donner les hypothèses éventuelles et les spécifications de l'algorithme 2.

Exercice 3 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$.

Algorithme 3 : Un algorithme mystère

Entrée : Un tableau T de taille n
Hypothèse : ??
Sortie : ??

```
1 i ← 0 ;
2 tant que i < n faire
3     si i = 0 alors
4         i ← i + 1 ;
5     sinon si T[i − 1] ⩽ T[i] alors
6         i ← i + 1 ;
7     sinon
8         échanger T[i − 1] et T[i] ;
9         i ← i − 1 ;
```

Q. 1 Donner les spécifications de l'algorithme 3.

Q. 2 Prouver la correction de l'algorithme 3.

Exercice 4 : Réécriture de boucles

On considère l'algorithme ci-dessous.

Algorithme 4 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$, dont chaque case contient un tableau de taille $n \in \mathbb{N}$

```
1 C ← 0 ;
2 pour i = 0 à n − 1 faire
3     pour j = 0 à n − 1 faire
4         si i + j ≡ 0[2] alors
5             T[i][j] ← C ;
6             C ← C + 1 ;
```

Q. 1 Donner un programme équivalent, sans boucle **pour**, et sans conditionnel **si**.

Q. 2 Et avec au plus un **tant que** ?

Exercice 5 : Réécriture de boucles

On considère l'algorithme ci-dessous.

Algorithme 5 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$, dont chaque case contient un tableau de taille $n \in \mathbb{N}$

```
1 C ← 0 ;
2 pour i = 0 à n − 1 faire
3     pour j = 0 à n − 1 faire
4         si i + j = n alors
5             T[i][j] ← C ;
6             C ← C + 1 ;
```

Q. 1 Donner un programme équivalent, sans boucle **pour**, et sans conditionnel **si**.

Q. 2 Et avec au plus un **tant que** ?

Exercice 6 : Réécriture de boucles

On considère l'algorithme ci-dessous.

Algorithme 6 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$, dont chaque case contient un tableau de taille $n \in \mathbb{N}$

```
1 C ← 0 ;
2 pour d = 0 à n − 1 faire
3     pour i = 0 à n − 1 − d faire
4         T[i][d + i] ← C ;
5         C ← C + 1 ;
```

Q. 1 Donner un programme équivalent, sans boucle **pour**.

Q. 2 Et avec au plus un **tant que** ?

Exercice 7 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$. On considère l'algorithme ci-dessous, dans lequel on a laissé un trou $\square$. $a//b$ désigne le quotient de la division euclidienne de a par b.

Algorithme 7 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$
Hypothèse : ??
Sortie : ??

```
1  K ← 0 ;
2  δ ← 0 ;
3  S ← 0 ;
4  pour i = 0 à □ faire
5      si i ≡ 0[2] alors
6          δ ← 1 ;
7          S ← (n − (i//2) − 1) ;
8      sinon
9          δ ← −1 ;
10         S ← i//2 ;
11     tant que K ≠ S faire
12         Échanger le contenu de T[K] et T[K + δ] ;
13         K ← K + δ ;
14     K ← K − δ ;
```

Q. 1 Suivant les valeurs choisies pour le trou $\square$, décrire le comportement de l'algorithme 8. *On s'attend à une disjonction de cas : "Dans le cas où $\square < 0$, l'algorithme calcule …, dans le cas où $\square = 42$, l'algorithme effectue une erreur de division par zéro, etc ….".*

Q. 2 Modifier l'algorithme précédent de sorte qu'il ne contienne plus de conditionnelle **si …alors …sinon**. Vous devez fournir un algorithme qui a exactement le même comportement au sens où les échanges de contenu se déroulent dans le même ordre.

Exercice 8 : Un algorithme mystère

On considère la fonction Mystere ci-dessous.

```
1  Procedure Mystere(n) :
2      si n > 100 alors
3          retourner n − 10 ;
4      sinon
5          retourner Mystere(Mystere(n + 11)) ;
```

Q. 1 Décrire le comportement de l'algorithme Mystere.

Q. 2 Donner un algorithme, non récursif, ayant le même comportement que la fonction Mystere.

Exercice 9 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$. On considère l'algorithme ci-dessous, dans lequel on a laissé un trou $\square$. $a//b$ désigne le quotient de la division euclidienne de a par b.

Algorithme 8 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$
Hypothèse : ??
Sortie : ??

```
1 δ ← 1 ;
2 pour i = 0 à □ faire
3     pour k = 0 à n//(2δ) − 1 faire
4         si T[2kδ] < T[(2k + 1)δ] alors
5             Échanger le contenu de T[2kδ] et T[(2k + 1)δ] ;
6     δ ← 2δ ;
7 retourner T[0]
```

Q. 1 Suivant les valeurs choisies pour le trou $\square$, décrire le comportement de l'algorithme 8. On n'hésitera pas à contraindre les entrées si on le juge pertinent. *On s'attend à une disjonction de cas : "Dans le cas où $\square < 0$, l'algorithme calcule …, dans le cas où $\square = 42$, l'algorithme effectue une erreur de division par zéro, etc ….".*

Q. 2 Transformer l'algorithme précédent en algorithme récursif. On impose que les échanges aient lieu dans le même ordre.

Exercice 10 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$.

Algorithme 9 : Un algorithme mystère

```
   Entrée : T un tableau de taille n ∈ ℕ
   Hypothèse : ??
   Sortie : ??
 1 Procedure SMystereA(i) :
 2     si i = n − 1 alors
 3         retourner T[i] ;
 4     sinon
 5         x ← SMystereA(i+1) ;
 6         si T[x] > T[i] alors
 7             retourner x ;
 8         sinon
 9             retourner i ;
10 Procedure SMystereB(i) :
11     si i = n − 1 alors
12         retourner T[i] ;
13     sinon
14         x ← SMystereB(i+1) ;
15         si T[x] < T[i] alors
16             retourner x ;
17         sinon
18             retourner i ;
19 pour i = 0 à n − 1 faire
20     si i ≡ 0[2] alors
21         Échanger le contenu de T[i] et T[SMystereA(i)] ;
22     sinon
23         Échanger le contenu de T[i] et T[SMystereB(i)] ;
```

Q. 1 Donner les hypothèses éventuelles et les spécifications de l'algorithme 9.

Q. 2 Proposer un algorithme le plus proche possible de l'algorithme 9 (les opérations d'échange doivent avoir lieu dans le même ordre) mais n'utilisant pas de récursivité.

Q. 3 Implémenter cet algorithme en C.

Exercice 11 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$. On note V le booléen vrai et F le booléen faux.

Algorithme 10 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$
Hypothèse : ??
Sortie : ??

```
1  Soit A un tableau de taille n ∈ ℕ, dont chaque case contient un tableau de taille n ∈ ℕ,
     initialisé à F.
2  pour i = 0 à n − 1 faire
3      A[i][T[j]] ← V ;
4  pour j = 0 à n − 1 faire
5      R ← F ;
6      pour i = 0 à n − 1 faire
7          si A[i][j] alors
8              R ← V
9      si R = F alors
10         retourner F ;
11 retourner V ;
```

Q. 1 Donner les hypothèses éventuelles et les spécifications de l'algorithme 10.

Q. 2 Proposer un algorithme le plus proche possible de l'algorithme 10 (même allocation d'un tableau de tableau) mais n'utilisant pas d'instruction **retourner** dans le corps de boucle.

Q. 3 Implémenter cet algorithme en C, dans le cas où les tableaux sont de taille 9.

Exercice 12 : Un algorithme mystère

Dans cet exercice les tableaux de taille $n \in \mathbb{N}$ sont indicés par des entiers de $[\![0, n-1]\!]$. On note V le booléen vrai et F le booléen faux.

Algorithme 11 : Un algorithme mystère

Entrée : $n \in \mathbb{N}$
Hypothèse : ??
Sortie : ??

```
1  Soit T un très grand tableau de taille M initialisé à 0 ;
2  Procedure SMystereA(b) :
3      si b alors
4          I ← −1 ; D ← M − 1 ; F ← 0 ;
5      sinon
6          I ← 1 ; D ← 0 ; F ← M − 1 ;
7      C ← D ;
8      tant que C ≠ F faire
9          T[C] ← T[C + I] ;
10         C ← C + I ;
11 Procedure SMystereB() :
12     si T[0] = T[1] alors
13         T[1] ← T[0] + T[1] ;
14         retourner V
15     sinon
16         retourner F
17 pour i = 0 à 2^n − 1 faire
18     SMystereA(F) ;
19     T[0] ← 1 ;
20     tant que SMystereB() faire
21         SMystereA(V) ;
22 retourner T[0]
```

Q. 1 Donner les hypothèses éventuelles et les spécifications de l'algorithme 43.

Q. 2 Implémenter cet algorithme en C.

Exercice 13 : Réécriture de boucles

On considère l'algorithme ci-dessous.

Algorithme 12 : Un algorithme mystère

Entrée : T un tableau de taille $n \in \mathbb{N}$, dont chaque case contient un tableau de taille $n \in \mathbb{N}$

```
1  C ← 0 ;
2  pour d = 0 à n − 1 faire
3      pour i = 0 à n − 1 − d faire
4          T[i][d + i] ← C ;
5          C ← C + 1 ;
```

Q. 1 Donner un programme équivalent, sans boucle **pour**.

Q. 2 Et avec au plus un **tant que** ?

Exercice 14 : Tableaux 1

Q. 1 Donner une fonction C prenant en argument un tableau d'entiers et le modifiant pour qu'il contienne la somme cumulée (la i-ème case contient la somme des i premières cases) du tableau initial.

Exercice 15 : Tableaux 2

Q. 1 Donner une fonction C prenant en argument un tableau d'entiers et le modifiant de la manière suivante : la première case est inchangée, les cases i suivantes reçoivent la différence initiale entre la case i et la case $i-1$. Par exemple `{1, 2, 5, 1}` est transformé en `{1, 1, 3, -4}`.

Exercice 16 : Tableaux 3

Q. 1 Donner une fonction C prenant en argument un tableau d'entiers et le lissant de la manière suivante : chaque case extrême est non modifiée, les cases internes reçoivent la moyenne des deux cases voisines.

Exercice 17 : Structures 1

Q. 1 Définir un type structuré `date` permettant la représentation d'une date contenant les grandeurs année, mois, jour.

Q. 2 Définir une fonction `inf_date` prenant deux dates en arguments et testant si la première précède la seconde au sens large.

Q. 3 Définir un type structuré `personne` permettant la représentation d'une personne ayant : un nom, une date de naissance et une date de décès.

Q. 4 Définir une fonction `en_vie` prenant en arguments une date et une personne et testant si cette personne est en vie à cette date.

Q. 5 Définir une fonction `rencontre_possible` prenant en arguments deux personnes et testant s'il est possible que ces deux personnes se soient rencontrées.

Q. 6 Définir une fonction `nb_en_vie` prenant en arguments un tableau de personnes et une date et retournant le nombre de personnes dans le tableau qui sont en vie à la date indiquée.

Exercice 18 : Structures 2

Q. 1 Définir un type structuré `points` permettant la représentation d'un point du plan.

Q. 2 Définir un type structuré `rectangle_par` permettant la représentation d'un rectangle dont les côtés sont parallèles aux axes du plan.

Q. 3 Définir une fonction `dans_rectangle` prenant en arguments un point et un rectangle et testant si le point est dans le rectangle.

Q. 4 Définir une fonction `rectangle_intersecte` prenant en arguments deux rectangles et testant s'ils s'intersectent.

Q. 5 Définir une fonction `rectangle_intersection` prenant en arguments deux rectangles qui s'intersectent et calculant l'intersection (un rectangle).

Q. 6 Définir une fonction `translate` prenant en arguments un rectangle et un point P et retournant le rectangle obtenu par translation du rectangle d'entrée du vecteur $\overrightarrow{OP}$.

Exercice 19 : Structures 3

Q. 1 Définir un type structuré `intervalle` permettant la représentation d'un intervalle $[\![a; b]\!]$ de $\mathbb{Z}$ avec $a, b \in \mathbb{Z}^2$.

Q. 2 Définir une fonction `intervalle_intersecte` permettant de tester si deux intervalles s'intersectent.

Q. 3 Définir une fonction `intervalle_intersection` permettant le calcul de l'intersection de deux intervalles.

Q. 4 Définir une fonction `intervalle_union` permettant le calcul de l'union de deux intervalles d'intersection non nulle.

Q. 5 Définir un type structuré `sous_ensembles_z` permettant la représentation d'un ensemble d'entiers. Votre structure contiendra un tableau T de 100 intervalles et un entier x. Ce type structuré représente alors l'ensemble $\bigcup_{i=0}^{x-1} \texttt{T[i]}$.

Q. 6 Définir une constante `vide` représentant l'ensemble vide.

Q. 7 Définir une fonction `ajout` permettant l'ajout d'un intervalle à un ensemble d'entiers. Si l'ensemble d'entiers contient déjà un intervalle qui s'intersecte avec l'intervalle à ajouter, on fera l'union de ces deux intervalles. Sinon on en ajoutera un nouveau.

Exercice 20 : Mémoire 1

```c
void g(int **p, int n) {
    **p = n; ②
    *p = &(**p) + 1; ③
}

void f(int* p) {
    g(&p, 0); ④
    g(&p, 1); ⑤
    g(&p, 2);
}

int main() {
    int tab[3]; ①
    f(&tab[0]);
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 21 : Mémoire 2

```c
int* choose(int* p1, int* p2) {
  bool b = rand() % 2 == 0;
  int* res;  ①
  if (b) {
    res = p1;
  } else {
```

```
    res = p2;
  }  ②
  return res;
}

void f(int tab[], int tab_len) {
  int* p =choose(&tab[0], &tab[tab_len-1]);    ③
  *p = 0;
}

int main() {
  int tab[3] = {1, 2, 3}; ⓪
  f(tab, 3); ④
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 22 : Mémoire 3

```
void g(int* p1, int* p2, int ** p) {
  ①
  if (*p1 > *p2) {
    *p = p1;
  } else {
    *p = p2;
  }
}

void f(int tab[4], int* ptab[2], int** p) {
  g(&tab[0], &tab[1], &ptab[0]); ②
  g(&tab[2], &tab[3], &ptab[1]); ③
  g(ptab[0], ptab[1], p); ④
}

int main() {
  int tab[4] = {0, 3, 2, 1};
  int* ptab[2];
  int* p;    ⓪
  f(tab, ptab, &p); ⑤

}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 23 : Mémoire 4

```c
typedef struct {
    int* p;
    int  x;
} myst;

myst g(myst m) {
    m.p = &m.x; ④
    return m;
}

myst f(myst m) {
    *m.p = *m.p + 1;
    m.x = 3; ②
    return m;
}

int main() {
    int x0 = 0;
    myst tt1 = {&x0, x0};
    myst tt2 = {&x0, x0}; ①
    tt2 = f(tt1); ③
    tt1 = g(tt1); ⑤
    *tt1.p = 2; ⑥
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 24 : Mémoire 5

```c
typedef struct {
    int* p1;
    int* p2;
    int  x;
} myst;

myst g(myst* pm, myst m) {
    m.x = *(m.p1);
    *((*pm).p1) = 4;    ②
    (*pm).p2 = &(m.x); ③
    return m ;
}

int main() {
    int t[3] = {1, 5, 6};
    myst m1 = {&t[0], &t[1], t[2]};
    myst m2 = {&t[1], &t[0], t[2]}; ①
    m2 = g(&m1, m2); ④
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 25 : Mémoire 6

```c
typedef struct {
    int** pp;
    int*  p;
    int   x;
} myst;

void f(myst* m1, myst* m2, myst m3) {
    (*m1).pp = &m3.p;
    (*m2).pp = &m3.p; ②
    m3.p = &((*m1).x); ③
    (*m1).p = &((*m2).x);
    (*m2).p = &((*m1).x); ④
    return;
}

int main() {
    int x  = 0;
    int *p = &x;
    myst m1 = {&p, p, x};
    myst m2 = {&p, p, x}; ①
    f(&m1, &m2, m1); ⑤
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 26 : Mémoire 7

```c
int** f() {
    int* tab[4];
    return &tab;
}

int** g() {
    int** tab = (int**) malloc(4 * sizeof(int*));
    for (int i = 0 ; i < 2; i++) {
        tab[i] = (int*) &(tab[i+1]);
    }②
    tab[3] = (int*) &(tab[0]);
    return tab;
}

int main() {
    int** y = f() ; ①
    int** x = g() ; ③
    int   t = 3;
    *((int**) *((int**) *x)) = &t; ④
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 27 : Mémoire 8

```c
typedef struct {
    int* a;
    int* b;
} s1 ;

typedef struct {
    int len;
    s1* x;
} s2 ;

s1* make_s1(int len) {
    s1* res = malloc(sizeof(s1));
    res->a  = malloc(len*sizeof(int));
    res->b  = malloc(len*sizeof(int)); ①
    return res;
}

s2* make_s2(int len) {
    s1* x    = make_s1(len);
    s2* res  = malloc(sizeof(s2));
    res->len = len;
    res->x   = x; ②
    return res;
}

int main() {
    s2* ss   = make_s2(4); ③
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 28 : Mémoire 9

```c
typedef struct {
    int* l;
    int* r;
    int* ctnu;
} s1;

s1* make(int len) {
    s1* res = (s1*) malloc(sizeof(s1));
    res->ctnu=malloc(len*sizeof(len));
    res->l=&(res->ctnu[1]);
    res->r=&(res->ctnu[len-2]); ①
    return res;
}

int main() {
    s1* m = make(4); ②
    m->ctnu[0]=3;
    m->l[0]=2; ③
    m->l[1]=1;
    m->r[1]=0; ④
}
```

Q. 1 Représenter la mémoire aux points de programme marqués ci-dessus.

Exercice 29 : Mémoire 10

```c
#include <stdlib.h>
#include <stdio.h>

void g(int*** p, int n) {
    if (n != 0) {
        (*p)--;
        ***p = n;
        g(p, n-1);
    }
}

int f(int*** p, int n, int m) {
    int q = 0;
    if (n != 0) {
        **p = &q;
        (*p)++;
        return f(p, n-1, m+1) * q;
    } else {
        g(p, m);
        return 1;
    }
}

int main() {
    int n = 5;
    int** p = malloc(sizeof(int*) * 5);
    printf("%d\n", f(&p, 5, 0));
}
```

Q. 1 En représentant la mémoire à des points de programmes bien choisis, décrire le comportement du programme ci-dessus.

Exercice 30 : Mémoire 11

```c
struct m {
    struct m* p1;               /* un pointeur ... */
    struct m* p2;               /* ... et un autre */
};
typedef struct m s;

void f(int n, s* p) {
    /* une première fonction mystere */
    if (n != 0) {
        p->p1 = p-1;
        p->p2 = p-2;
        f(n-1, p+1);
        return ;
    }
}

int g(s p) {
    /* une seconde */
    if (p.p1 != NULL && p.p2 != NULL) {
        return g(*(p.p1)) + g(*(p.p2));
    } else {
        int cpt = 0;
        if (p.p1 != NULL) {cpt ++;}
        if (p.p2 != NULL) {cpt ++;}
        return cpt;
    }
}

int mystere(int n) {
    /* et une troisième qui s'appelle mystere */
    assert(n >= 2);
    s* p = (s*) malloc((n+1) * sizeof(struct m));
    p[0].p1 = NULL;
    p[0].p2 = NULL;
    p[1].p1 = &p[0];
    p[1].p2 = NULL;
    f(n-1, &p[2]);
    return g(p[n]);

}

int main() {
    mystere(5);
}
```

Q. 1 En représentant la mémoire à des points de programmes bien choisis, décrire le comportement du programme ci-dessus.

Exercice 31 : Mémoire 12

```c
int s(int m, int** tp) {
    int r = 0;
    for (int i = 0 ; i < m ; i ++) {
        if (tp[i] != NULL) {
            r += *tp[i];
        }

    }
    return r;
}

int f(int n, int m, int** tp, int** tpi) {
    if (n != 0) {
        if (n % 2 == 0) {
            int mem = 1;
            *tp = &mem;
            return f(n-1, m, tp+1, tpi);
        } else {
            int* pmem = (int*) malloc(sizeof(int) * 1);
            *pmem = 1;
            *tp = pmem;
            return f(n-1, m, tp+1, tpi);
        }
    } else {
        return s(m, tpi);
    }
}

int main() {
    int n = 6;
    int** tp = malloc(sizeof(int*) * n);
    return f(n, n, tp, tp) + s(n, tp);
}
```

Q. 1 En représentant la mémoire à des points de programmes bien choisis, décrire le comportement du programme ci-dessus.

Exercice 32 : Mémoire 13

```c
struct liste_s {                /* Définition d'un type liste */
    struct liste_s* next;
} ;
typedef struct liste_s* liste ;
liste myst1() {/* ... ??? */
    liste res = (liste) malloc(sizeof(struct liste_s));
    res->next = res;
    return res;
}
liste myst2(liste l1) {/* ... ??? */
    liste cur = l1;
    while (cur->next != l1) {
        cur = cur->next;
    }
    return cur;
}
liste myst3(liste l1, liste l2) {/* ... ??? */
    liste f1 = myst2(l1);
    liste f2 = myst2(l2);
    f1->next=l2;
    f2->next=l1;
    return l1;
}
int myst4(liste l1) {/* ... ??? */
    liste cur = l1;
    int res = 1;
    while (cur->next != l1) {
        cur = cur->next;
        res ++;
    }
    return res;
}
liste myst5(liste* t, int g, int d) {
    if (g == d) {return t[g];}
    else {
        int mid = (g+d)/2 ;
        liste l1 = myst5(t, g, mid);
        liste l2 = myst5(t, mid+1, d);
        return myst3(l1, l2);
    }
}
int g(int n) {/* ... ??? */
    liste* t = (liste*) malloc(sizeof(liste) * n);
    for (int i = 0 ; i < n ; i ++) { t[i] = myst1(); }
    liste l = myst5(t, 0, n-1);
    return myst4(l);
}
int main() {printf("%d", g(42));}
```

Q. 1 En représentant la mémoire à des points de programmes bien choisis, décrire le comportement du programme ci-dessus.

Exercice 33 : Mémoire 14

```c
#include <stdlib.h>
#include <stdio.h>
void print(char*);

struct liste_s {                /* Définition d'un type liste */
    int    n;
    struct liste_s* next;
} ;
typedef struct liste_s* liste ;

liste f(liste p) {
    /* ... ??? */
    liste res = (liste) malloc(sizeof(struct liste_s));
    res->next = p;
    res->n    = 0;
    return res;
}
int pow2(int n) {
    /* Calcule 2^n */
    return 1 << n;
}

void h(liste p) {
    /* ... ??? */
    liste cur;
    for (cur = p; cur != NULL; cur = cur->next) {
        cur->n ++;
    }
}

int g(int n) {
    /* ... ??? */
    liste* t = (liste*) malloc(sizeof(liste) * pow2(n));
    liste x = f(NULL);
    t[0] = x;
    for (int i = 1; i < n+1; i++) {
        for (int j = pow2(i)-1; j >= 0; j = j - 2) {
            t[j  ] = f(t[(j-1)/2]);
            t[j-1] = f(t[(j-1)/2]);
        }
    }
    for (int j = 0; j < pow2(n) ; j ++) {
        h(t[j]);
    }
    return (x->n);
}

int main() {printf("%d", g(4));}
```

Q. 1 En représentant la mémoire à des points de programmes bien choisis, décrire le comportement du programme ci-dessus.

Exercice 34 : Correction d'algorithmes

Q. 1 Pour chacun des algorithmes suivants, expliquer ce que fait l'algorithme, prouver sa terminaison et sa correction.

Algorithme 13 : Mystère

Entrée : Deux entiers naturels i et j
Sortie : ??

```
1  I ← i ;
2  J ← j ;
3  K ← 0 ;
4  R ← 0 ;
5  tant que I > 0 ou J > 0 faire
6      si J = 0 alors
7          J ← K ;
8          K ← 0 ;
9          I ← I − 1 ;
10     sinon
11         J ← J − 1 ;
12         K ← K + 2 ;
13         R ← R + 1 ;
14 retourner R
```

Algorithme 14 : Mystère

Entrée : Un entier n qui est une puissance de 2 : $n = 2^k$
Sortie : ??

```
1  A ← 0 ;
2  I ← 1 ;
3  tant que I < n faire
4      J ← 0 ;
5      tant que J < n faire
6          A ← A + 1 ;
7          J ← J + 1 ;
8      I ← 2I ;
9  retourner A + 2
```

Algorithme 15 : Mystère

Entrée : Deux entiers l et u
Sortie : ??

```
1  L ← l ;
2  U ← u ;
3  C ← l ;
4  A ← 0 ;
5  tant que U − L > 0 faire
6      L ← L + 1 ;
7      tant que C < U faire
8          C ← C + 1 ;
9          A ← A + 1 ;
10     U ← U − 1 ;
11     tant que C > L faire
12         C ← C − 1 ;
13         A ← A + 1 ;
14     L ← L + 1 ;
15 retourner A
```

Algorithme 16 : Mystère

Entrée : Deux entiers naturels a et b
Sortie : ??

```
1  A ← a ;
2  B ← b ;
3  R ← 0 ;
4  tant que B > 0 faire
5      si B ≡ 1[2] alors
6          R ← A + R ;
7          B ← B − 1 ;
8      sinon
9          A ← 2A ;
10         B ← B/2 ;
11 retourner R
```

Algorithme 17 : Mystère

Entrée : Un entier naturel n
Sortie : ??

$r \leftarrow 1$;
$a \leftarrow 2$;
$k \leftarrow n$;
tant que $k > 0$ **faire**
  si $k \equiv 1[2]$ **alors**
    $r \leftarrow a \times r$
  $a \leftarrow a \times a$;
  $k \leftarrow k//2$;
retourner r

Algorithme 18 : Mystere

Entrée : Un entier naturel positif ou nul $n \geqslant 0$
Sortie : ??

$Q \leftarrow 0$;
$S \leftarrow 0$;
$I \leftarrow 0$;
tant que $Q \leqslant n$ **faire**
  $Q \leftarrow Q + 3S + 3I + 1$;
  $S \leftarrow S + 2I + 1$;
  $I \leftarrow I + 1$
retourner $I - 1$

Algorithme 19 : Mystere

Entrée : Deux entiers naturels a et b avec $a \neq 0$ et $b \neq 0$
Sortie : ??

$A \leftarrow a$;
$B \leftarrow b$;
tant que $A \neq B$ **faire**
  si $B < A$ **alors**
    $A \leftarrow A - B$;
  sinon
    $B \leftarrow B - A$;
retourner A

Algorithme 20 : Mystere

Entrée : Un entier naturel positif ou nul $n \geqslant 0$
Sortie : ??

$S \leftarrow 0$;
$I \leftarrow 0$;
tant que $S \leqslant n$ **faire**
  $S \leftarrow S + 2I + 1$;
  $I \leftarrow I + 1$
retourner $I - 1$

Algorithme 21 : Mystere

Entrée : Un entier naturel positif strictement $n > 0$
Sortie : ??

$P \leftarrow 1$;
$K \leftarrow 0$;
tant que $P \leqslant n$ **faire**
  $K \leftarrow K + 1$;
  $P \leftarrow P \times 2$;
retourner $K - 1$

Algorithme 22 : Mystere

Entrée : n un entier naturel, m un entier naturel non nul
Sortie : ??

$I \leftarrow n$;
tant que $I > m$ **faire**
  $I \leftarrow I - m$;
retourner $I \stackrel{?}{=} 0$

Algorithme 23 : Mystere

Entrée : Un entier naturel n
Sortie : ??

$I \leftarrow n$;
$J \leftarrow 0$;
tant que $I > 0$ **faire**
  $I \leftarrow I - 1$;
  $J \leftarrow J + 2$;
retourner J

Algorithme 24 : Mystere

Entrée : n un entier naturel, m un entier naturel
Sortie : ??

$I \leftarrow n$;
$J \leftarrow m$;
tant que $J > 0$ **faire**
  $I \leftarrow I + 1$;
  $J \leftarrow J - 1$;
retourner I

Algorithme 25 : Mystere

Entrée : Un entier naturel n
Sortie : ??

```
1 I ← 0 ;
2 B ← vrai ;
3 R ← 0 ;
4 tant que I < n faire
5     si B alors
6         R ← R + I ;
7     B ← non(B) ;
8     I ← I + 1 ;
9 retourner R
```

Algorithme 26 : Mystere

Entrée : n un entier naturel
Sortie : ??

```
1 I ← 0 ;
2 R ← 0 ;
3 tant que I < n faire
4     R ← R + I ;
5     I ← I + 2 ;
6 retourner R
```

Algorithme 27 : Mystere

Entrée : Un entier naturel n
Sortie : ??

```
1  I ← 0 ;
2  A ← 0 ;
3  B ← 0 ;
4  tant que I < n faire
5      A ← A + I ;
6      A ← A + B ;
7      B ← A − B ;
8      A ← A − B ;
9      I ← I + 1
10 retourner A
```

Algorithme 28 : Mystere

Entrée : n un entier naturel
Sortie : ??

```
1  I ← 1 ;
2  R ← 1 ;
3  tant que I < 2n + 1 faire
4      R ← R × I ;
5      si I < n alors
6          I ← 2n + 1 − I ;
7      sinon si I = n + 1 alors
8          I ← 2n + 1 ;
9      sinon
10         I ← 2n + 2 − I ;
11 retourner R
```

Exercice 35 : Correction d'algorithmes manipulant des tableaux

Q. 1 Expliquer ce que fait l'algorithme, prouver sa terminaison et sa correction.

Algorithme 29 : Mystère 1

Entrée : Un tableau T, non vide de taille n
Sortie : ??

```
1 M ← T[0] ;
2 I ← 0 ;
3 X ← 0 ;
4 tant que I < n faire
5     si T[I] > M alors
6         M ← T[I] ;
7         X ← I
8     I = I + 1
9 retourner X
```

Algorithme 30 : Mystère 2

Entrée : Un tableau T de taille n, un élément e
Sortie : ??

```
1 V ← faux ;
2 I ← 0 ;
3 tant que non(V) et I < n faire
4     si T[I] = e alors
5         V ← vrai
6     I = I + 1
7 retourner V
```

Algorithme 31 : Mystère 3

Entrée : Un tableau T de taille n
Sortie : ??

```
1 V ← vrai ;
2 I ← 0 ;
3 tant que V et I < n − 1 faire
4     si T[I] > T[I + 1] alors
5         V ← faux
6     I = I + 1
7 retourner V
```

Algorithme 32 : Mystère 4

Entrée : Un tableau T_1 de taille n_1, un tableau T_2 de taille n_2
Sortie : ??

```
1 I_1 ← 0 ;
2 I_2 ← 0 ;
3 tant que I_1 < n_1 et I_2 < n_2 faire
4     si T_1[I_1] = T_2[I_2] alors
5         I_1 = I_1 + 1 ;
6         I_2 = I_2 + 1 ;
7     sinon
8         I_2 = I_2 + 1 ;
9 retourner I_1 = n_1
```

Algorithme 33 : Mystère 5

Entrée : Un tableau T de taille n d'éléments tous disjoints, un élément e
Sortie : ??

```
1  I ← 0 ;
2  A ← 0 ;
3  B ← 0 ;
4  tant que I < n faire
5      si T[I] > e alors
6          A ← A + 1 ;
7      sinon si T[I] < e alors
8          B ← B + 1 ;
9      I = I + 1
10 retourner A = B
```

Algorithme 34 : Mystère 7

Entrée : Un tableau t de taille n d'éléments, indicés par $[\![0, n-1]\!]$.
Sortie : ??

```
1 I ← 0 ;
2 tant que I < n//2 et t[I] = t[n − I − 1] faire
3     I = I + 1
4 retourner I = n//2
```

Algorithme 35 : Mystère 8

Entrée : Un tableau t de taille n d'entiers, indicés par $[\![0, n-1]\!]$.
Sortie : ??

```
1 I ← 0 ;
2 tant que I < n et t[I] ≠ 0 faire
3     I = I + 1
4 retourner I = n
```

Algorithme 36 : Mystère 9

Entrée : Un tableau t de taille n d'entiers, indicés par $[\![0, n-1]\!]$.
Sortie : ??

$I \leftarrow 0$;
tant que $I < n-1$ et $t[I] = t[I+1]$ **faire**
  $I = I + 1$
retourner $I = n - 1$

Algorithme 37 : Mystère 10

Entrée : Un entier s, et un tableau t de taille n d'entiers naturels, indicés par $[\![0, n-1]\!]$.
Sortie : ??

$R \leftarrow s$;
$I \leftarrow 0$;
tant que $I < n-1$ et $R > 0$ **faire**
  $R \leftarrow R - t[I]$;
  $I \leftarrow I + 1$;
retourner $R \stackrel{?}{=} 0$ et $I \stackrel{?}{=} n$

Algorithme 38 : Mystère 11

Entrée : Un tableau t de taille n d'entiers, indicés par $[\![0, n-1]\!]$.
Sortie : ??

$B \leftarrow \mathsf{V}$;
$I \leftarrow 0$;
tant que $I < n$ **faire**
  si $T[I] = 0$ **alors**
    $B \leftarrow \mathsf{non}(B)$;
  $I \leftarrow I + 1$
retourner B

Algorithme 39 : Mystère 12

Entrée : Un tableau t de taille n d'entiers positifs ou nuls.
Sortie : ??

$AM \leftarrow \emptyset$;
$M \leftarrow -1$;
$I \leftarrow 0$;
tant que $I < n$ **faire**
  si $T[I] > M$ **alors**
    $M \leftarrow T[I]$;
    $AM \leftarrow \{I\}$;
  sinon
    $AM \leftarrow \{I\} \cup AM$;
  $I \leftarrow I + 1$;
retourner $\mathsf{card}(AM) \stackrel{?}{=} 1$

Algorithme 40 : Mystère 13

Entrée : Un tableau t de taille $n \geqslant 0$, $e \in \mathbb{N}$.
Sortie : ??

$I \leftarrow 0$;
$A \leftarrow 0$;
$B \leftarrow 0$;
tant que $I < n$ **faire**
  si $T[I] < e$ **alors**
    $A \leftarrow A + 1$;
  sinon si $T[I] = e$ **alors**
    $B \leftarrow B + 1$;
  $I \leftarrow I + 1$;
retourner $A \leqslant \frac{n}{2}$ et $B > \frac{n}{2}$

Exercice 36 : Correction d'algorithmes manipulant des tableaux

Q. 1 Pour chacun des algorithmes suivants, expliquer ce que fait l'algorithme, prouver sa terminaison et sa correction.

Algorithme 41 : Mystère

Entrée : Un tableau T de taille n d'éléments tous distincts
Sortie : ??

$M \leftarrow 0$;
$I \leftarrow 0$;
tant que $I < n/2$ **faire**
  $M \leftarrow T[I]$;
  $T[I] \leftarrow T[n-1-I]$;
  $T[n-I-1] \leftarrow M$;
  $I = I + 1$

Algorithme 42 : Mystère

Entrée : Un tableau T de taille n d'entiers
Sortie : ??

$I \leftarrow 1$;
tant que $I < n$ **faire**
  $T[I] \leftarrow T[I] + T[I-1]$;
  $I \leftarrow I + 1$
retourner V

Algorithme 43 : Mystère

Entrée : Un entier naturel n
Sortie : ??

Soit T un tableau de n entiers initialisés à 1 ;
$I \leftarrow 1$;
tant que $I < n$ **faire**
  $J \leftarrow 0$;
  tant que $J < I$ **faire**
    $T[I] \leftarrow T[I] + T[J]$;
    $J \leftarrow J + 1$;
  $I \leftarrow I + 1$;
retourner V

Exercice 37 : Expressions 1

Pour chacune des expressions suivantes, donner, si elle est correcte, son type et sa valeur.

1.
```
let (a, b) = (2, 3) in
(a + b, b)
```

2.
```
let a, b = true, false in
if b then 1 else 3.5
```

3.
```
(fun g x -> g x)
  ((fun a x -> x + a) 1)
  (1)
```

Exercice 38 : Expressions 2

Pour chacune des expressions suivantes, donner, si elle est correcte, son type et sa valeur.

1.
```
let y = (5, "true") in
let (a, b) = y in
b
```

2.
```
let a, b = true, false in
if b then 1 else if a then 3 else 2
```

3.
```
let a =
  (fun b ->
     if b then fun x -> (3 * x)
     else fun x -> (2 * x))
in
(a true 1) * (a false 1)
```

Exercice 39 : Expressions 3

Pour chacune des expressions suivantes, donner, si elle est correcte, son type et sa valeur.

1.
```
let x = ('a', 'b') in
let (a, b) = x in
a
```

2.
```
let a, b = true, "" in
if b then 1 else 2
```

3.
```
let a = (fun b ->
  if b then fun x -> (x+1)
  else fun x -> (x+2))
in
let t = (fun x -> x mod 2 = 0) in
(a (t 0) 0) + (a (t 1) 1)
```

Exercice 40 : Heures

On représente un *moment* dans la journée, par une paire d'entiers dénotant les heures et les minutes.

Q. 1 Définir une fonction OCaml `heure : int -> int -> (int * int)` permettant de fabriquer un moment à partir d'un nombre d'heures et d'un nombre de minutes.

Q. 2 Définir une fonction OCaml `apres : (int * int) -> (int * int) -> bool` permettant de tester si un moment m_1 si situe plus tard dans la journée qu'un moment m_2 au moyen de l'appel `apres` m_1 m_2.

Q. 3 Définir une fonction OCaml permettant de calculer le temps qui s'est écoulé, en minutes entre deux moments `ecoule : (int * int) -> (int * int) -> int`

Q. 4 Définir une fonction OCaml permettant de calculer si un moment x se trouve entre deux moments m_1, m_2.

Exercice 41 : Intervalles d'entiers

On représente un intervalle d'entier $[\![a, b]\!]$ en OCaml par une pair `(a, b)`.

Q. 1 Définir une fonction OCaml `singleton : int -> (int * int)` prenant en argument un entier et calculant l'intervalle contenant uniquement cet entier.

Q. 2 Définir une fonction OCaml `card: (int * int) -> int` prenant en argument un intervalle d'entiers et retournant son nombre d'éléments.

Q. 3 Définir une fonction OCaml `mem : int -> (int * int) -> bool` prenant un argument un entier x et un intervalle d'entier I et calculant si oui ou non $x \in I$.

Q. 4 Définir une fonction OCaml `intersection : (int * int) -> (int * int) -> (int * int)` prenant en argument deux intervalles et calculant leur intersection.

Exercice 42 : Multiplication

Dans cet exercice on ne s'autorise pas la multiplication d'entiers en OCaml

Q. 1 Proposer une mise en équation récursive de l'opération $a \times b$ où a et b sont des entiers naturels.

Q. 2 Proposer une fonction OCaml calculant $a \times b$ à partir des entiers a et b.

Exercice 43 : Addition

Dans cet exercice on ne s'autorise pas l'addition d'entiers en OCaml, seulement l'incrémentation/la décrémentation par 1.

Q. 1 Proposer une mise en équation récursive de l'opération $a+b$ où a et b sont des entiers naturels.

Q. 2 Proposer une fonction OCaml calculant $a + b$ à partir des entiers a et b.

Exercice 44 : Nombre de chiffres dans un nombre

Q. 1 Proposer une mise en équation récursive du nombre de chiffres dans un nombre n.

Q. 2 Proposer une fonction OCaml calculant le nombre de chiffres dans un nombre n passé en argument.

Exercice 45 : Ordre supérieur

Q. 1 Un sous-ensemble X d'un ensemble S peut être vu comme une fonction de S dans l'ensemble $\{0, 1\}$. Expliciter une telle fonction.

On représente donc un ensemble d'entiers en OCaml par une fonction de type `int -> bool`.

Q. 2 Définir une fonction `singleton : int -> (int -> bool)` calculant, étant donné un entier x, l'ensemble $\{x\}$.

Q. 3 Définir une fonction `mem : int -> (int -> bool) -> bool` prenant en argument un entier x et un ensemble d'entiers X et calculant vrai si et seulement si $x \in X$.

Q. 4 Définir une fonction `complementaire : (int -> bool) -> (int -> bool)` calculant le complémentaire d'un ensemble d'entier.

Q. 5 Définir une fonction `union : (int -> bool) -> (int -> bool) -> (int -> bool)` prenant en argument deux ensembles d'entiers et calculant leur union.

Exercice 46 : Pseudo Inverse 1

Q. 1 Écrire une fonction `inv_exp2: int -> int` telle que `(inv_exp2 n)` renvoie le plus grand $d \in \mathbb{N}$ tel que $2^d \leqslant$ `n`. On supposera que `n` $\geqslant 0$.

Q. 2 Quelle est la complexité de votre fonction ?

Exercice 47 : Pseudo Inverse 2

Q. 1 Écrire une fonction `inv_square: int -> int` telle que `(inv_square n)` renvoie le plus grand $d \in \mathbb{N}$ tel que $d^2 \leqslant$ `n`. On supposera que `n` $\geqslant 0$.

Q. 2 Quelle est la complexité de votre fonction ?

Exercice 48 : Pseudo Inverse 3

Q. 1 Écrire une fonction `inv_powpow: int -> int` telle que `(inv_powpow n)` renvoie le plus grand $d \in \mathbb{N}$ tel que $2^{2^d} \leqslant$ `n`. On supposera que `n` $\geqslant 0$.

Q. 2 Quelle est la complexité de votre fonction ?

Exercice 49 : Image des triplets

Q. 1 Donner une fonction `sum_img_triple` prenant en arguments une fonction f, un entier n et calculant $\sum_{(i,j,k)\in[\![0,n-1]\!]^3} f(i,j,k)$. On précisera le type d'une telle fonction

Q. 2 En déduire une fonction prenant en argument un entier n et calculant $\sum_{i=0}^{n-1} i^3$.

Q. 3 Généraliser la fonction `sum_img_triple` pour pouvoir calculer autre chose qu'une somme des $f(i,j,k)$. Par exemple, le produit, ou la concaténation.

Q. 4 Votre fonction `sum_img_triple` est elle récursive terminale ? Si non la rendre récursive terminale.

Exercice 50 : Image des pairs

Q. 1 Donner une fonction `sum_img_triple` prenant en arguments une fonction f, un entier n et calculant $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} f(i,j)$. On précisera le type d'une telle fonction

Q. 2 En déduire une fonction prenant en argument un entier n et calculant $\sum_{i=0}^{n-1} i^2$.

Q. 3 Généraliser la fonction `sum_img_pair` pour pouvoir calculer autre chose qu'une somme des $f(i,j)$. Par exemple, le produit, ou la concaténation.

Q. 4 Votre fonction `sum_img_pair` est elle récursive terminale ? Si non la rendre récursive terminale.

Exercice 51 : Image d'une progression géométrique

Q. 1 Donner une fonction `sum_img_prog` prenant en arguments une fonction f, un entier x, un entier n et calculant $\sum_{i=0}^{n-1} f(x^i)$. On précisera le type d'une telle fonction.

Q. 2 En déduire une fonction prenant en argument un entier n et calculant $\sum_{i=0}^{n-1} 2^{2i}$.

Q. 3 Généraliser la fonction `sum_img_prog` pour pouvoir calculer autre chose qu'une somme des $f(x)$. Par exemple, le produit, ou la concaténation.

Q. 4 Votre fonction `sum_img_prog` est elle récursive terminale ? Si non la rendre récursive terminale.

Exercice 52 : Polymorphisme 1

Q. 1 Donner une fonction `applique3` prenant en argument un triplet de fonction f, g, h et un triplet de valeurs x, y, z et retournant le triplet des images $f(x), f(y), f(z)$. Quel est le type le plus général de la fonction `applique3`.

Exercice 53 : Polymorphisme 2

Q. 1 Donner une fonction `tri2` prenant en arguments deux fonctions f et g et une valeur x et retournant la paire (f, g) si $f(x) < g(x)$ et la paire (g, f) sinon. Quel est le type le plus général de la fonction `tri2`.

Exercice 54 : Polymorphisme 3

Q. 1 Donner une fonction `comp_if` prenant en arguments trois fonctions f, g et h et un booléen b et retournant $f \circ g$ si b est vrai et $f \circ h$ sinon. Quel est le type de votre fonction ?

Exercice 55 : Entier retourné

Q. 1 Définir une fonction `fst_digit : int -> int * int` prenant en argument un entier naturel et retournant une paire telle que si `fst_digit z` $= (x, y)$ alors x est le premier chiffre de l'écriture de z en base 10 et y est le reste des chiffres de cette écriture. Par exemple `fst_digit 1234 = (1, 234)`.

Q. 2 En déduire une fonction `prepend : int -> int -> int` prenant en arguments un entier naturel x de $[\![0, 9]\!]$ et un entier y et retournant le nombre obtenu en plaçant le digit x en tête du nombre y. Par exemple `prepend 1 234 = 1234`

Q. 3 En déduire une fonction `rev: int -> int` retournant un entier naturel. Par exemple `rev 1234 = 4321`.

Q. 4 Donner une version récursive terminale de `rev`.

Exercice 56 : Entier divisible par 9

Q. 1 Écrire une fonction `sum_digits : int -> int` calculant la somme des digits d'un entier naturel.

Q. 2 Sans utiliser la fonction `mod` et sans la redéfinir écrire une fonction `div9 : int -> bool` calculant si un nombre est ou non un multiple de 9.

Exercice 57 : Puissance de 2 moins 1

Q. 1 Donner une fonction `forall` prenant en arguments une fonction f et un entier n. `forall` calcule si la fonction f, appliquée sur chacun des bits de l'écriture en base 2 de n vaut vrai. Un bit sera représenté au moyen d'un booléen `true` ou `false`. Donner le type de votre fonction `forall`.

Q. 2 En déduire une fonction `pow2m1 : int -> int` permettant de tester si un nombre est de la forme $2^p - 1$.

Exercice 58 : Entiers palindromes

Q. 1 Écrire une fonction permettant de tester si un entier est un palindrome.

Exercice 59 : Nombre d'occurrences dans une liste

Q. 1 Donner une fonction permettant le calcul du nombre d'occurrences d'un élément dans une liste.

Exercice 60 : Nombre d'occurrences du minimum

Q. 1 Donner une fonction permettant le calcul, à partir d'une liste l de la paire (x, nb) où x est le plus petit élément de l et nb est son nombre d'occurrence dans l.

Exercice 61 : Première occurrence satisfaisant un prédicat

Q. 1 Donner une fonction prenant en argument une fonction f et une liste l et calculant x le premier élément de l tel que $f(x)$ est vrai.

Exercice 62 : Position des occurrences

Q. 1 Définir une fonction prenant en paramètres une liste l et un élément x et donnant la liste des indices auxquels x se trouve dans la liste l.

Exercice 63 : Sous-séquence

Q. 1 Définir une fonction calculant, à partir d'une liste l, la plus longue sous liste d'éléments consécutifs identiques dans l. Votre fonction calculera un triplet : indice de départ de la sous-séquence en question, longueur de la sous-séquence en question, valeur des éléments de la sous-séquence en question.

Exercice 64 : Suppression du n-ième élément d'une liste

Q. 1 Définir une fonction prenant en argument un entier n et une liste l et calculant la liste l privée de son n-ième élément.

Exercice 65 : Regroupement en une liste d'association

Q. 1 Définir une fonction prenant en argument une liste l et calculant une liste de paires représentant dont le premier élément de la paire est un élément de l et le second son nombre d'occurrences dans l.

Exemples :

```
regroupe [1; 2; 3; 2; 4; 3] = [(1, 1); (2, 2); (3, 2); (4, 1)]
```

Exercice 66 : Deuxième plus grand

Q. 1 Donner une fonction donnant le deuxième plus grand élément d'une liste.

Exercice 67 : Un sur deux

Q. 1 Donner une fonction prenant en argument une liste l et calculant la liste obtenue en prenant un élément sur deux de l.

Exercice 68 : Représentation d'une fonction au moyen d'une paire de liste

On représente une fonction par deux listes de même tailles : la liste de ses antécédents, la liste de ses images.

Exemples :

`([1; 3; 2], [5; 5; 7])` représente la fonction $(1 \mapsto 5, 2 \mapsto 7, 3 \mapsto 5)$.

Q. 1 Définir une fonction `repr` prenant en argument une liste l et une fonction f et calculant la représentation par paire de listes de la fonction f sur l'espace de définition formé par l. On proposera un type pour cette fonction avant de la définir.

Q. 2 Définir une fonction `derepr` prenant en argument une liste l et une fonction f et calculant la fonction représentée par la liste. La fonction ainsi calculée devra lever une exception si la liste ne précise pas son comportement.

Exercice 69 : Fonctionnelle 1

Q. 1 Définir le type et une implémentation de la fonction `list_filter` prenant en paramètres une fonction f à valeurs booléennes et une liste l et retournant la liste des éléments x de l tels que $f(x)$ vaut vrai (dans l'ordre dans lequel ils apparaissent dans l).

Exercice 70 : Fonctionnelle 2

Q. 1 Définir le type et une implémentation de la fonction `list_map` prenant en paramètres une fonction f à valeurs booléennes et une liste l et retournant la liste des éléments $f(x)$ pour $x \in l$ (dans l'ordre dans lequel ils apparaissent dans l).

Exercice 71 : Anagramme

Donner une fonction permettant de tester si deux listes sont l'anagramme l'une de l'autre.

Exercice 72 : Énumération des listes de booléens

Q. 1 Donner une fonction `generate : int -> bool list list` prenant en argument un entier n et générant la liste de toutes les listes de booléens de taille n.

Q. 2 Donner une fonction `enumerate : int -> (int -> bool list)` prenant en argument un entier n et calculant une fonction f réalisant une bijection de $[\![0, 2^n - 1]\!]$ dans l'ensemble des listes de booléens de longueur n.

Exercice 73 : Liste 1

Q. 1 Implanter la fonction `partition: ('a -> bool) -> 'a list -> 'a list * 'a list` prenant en argument un prédicat et une liste et calculant la paire de la liste des éléments satisfaisant le prédicat et ceux ne le satisfaisant pas.

On définit le type `type 'a option = None | Some of 'a`

Q. 2 Implanter une fonction `filter_map : ('a -> 'b option) -> 'a list -> 'b list` telle que `filter_map f l` applique `f` à chaque élément de `l`, ôte de la liste les éléments pour lesquels l'appel produit `None` et calcule la liste des éléments se trouvant dans le `Some` sinon.

Exercice 74 : Liste 2

Q. 1 Implanter une fonction `combine : 'a list -> 'b list -> ('a * 'b) list` prenant en argument deux listes de même longueur et les combinant : `combine [a1; ...; an] [b1; ...; bn] = [(a1,b1); ...; (an,bn)]`.

Q. 2 Implanter une fonction `fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a` telle que `fold_left2 f init [a1; ...; an] [b1; ...; bn] = f (... (f (f init a1 b1) a2 b2) ...) an bn`

Exercice 75 : Liste 3

Q. 1 Définir une fonction `compare_l : 'a list -> 'a list -> int` comparant les deux listes pour l'ordre lexicographique (l'ordre du dictionnaire). Ainsi `compare_l l1 l2 = -1` si `l1` $<$ `l2`, `compare_l l1 l2 = 1` si `l1` $>$ `l2` et finalement `compare_l l1 l2 = 0` si `l1` $=$ `l2`.

Q. 2 Comment rendre cette fonction paramétrique en l'opérateur de comparaison des éléments dans les listes ?

Q. 3 Définir une fonction `map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` telle que `map2 f [a1; ...; an] [b1; ...; bn] = [f a1 b1; ...; f an bn]`.

Exercice 76 : Mélange

Dans cet exercice, on s'autorise à utiliser la fonction `List.rev : 'a list -> 'a list` calculant le miroir d'une liste.

Q. 1 Définir une fonction `melange: 'a list -> 'a list` prenant en paramètre une liste `[`x_0`; `x_1`; `x_2`; …; `x_{n-2}`; `x_{n-1}`]` et retournant la liste `[`x_0`; `x_{n-1}`; `x_1`; `x_{n-2}`; `x_2`; …]`

Q. 2 Si la réponse à la première question est de complexité quadratique en la taille de la liste initiale, même question mais avec une complexité linéaire.

Exercice 77 : Sous-listes continues

Q. 1 Écrire une fonction `ss_listes_continues: 'a list -> 'a list list` prenant en paramètre une liste `l` et calculant l'ensemble de ses sous-listes non vides d'éléments contigus dans la liste `l`.

```
ss_listes_continues [1; 2; 3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]; [2]; [2; 3]; [3]]
```

Exercice 78 : Produit cartésien

Q. 1 Écrire une fonction `produit_cartesien : 'a list -> 'b list -> ('a * 'b) list` prenant en paramètres une liste `[`x_0`; `x_1`; `x_2`; …; `x_{n-2}`; `x_{n-1}`]` et une liste `[`y_0`; `y_1`; `x_2`; …; `y_{m-2}`; `y_{m-1}`]` et retournant la liste `[(`x_0`, `y_0`); (`x_0`, `y_1`); …; (`x_0`, `y_{m-1}`); (`x_1`, `y_0`); (`x_1`, `y_1`); …; (`x_1`, `y_{m-1}`); …; (`x_{n-1}`, `y_0`); (`x_{n-1}`, `y_1`); …; (`x_{n-1}`, `y_{m-1}`);]`.

Exercice 79 : Suppression des occurrences consécutives

Q. 1 Écrire une fonction `compresse : 'a list -> 'a list` qui élimine les éléments consécutifs identiques d'une liste. Par exemple :

```
# compresse [1; 1; 2; 3; 3; 3; 4; 4];;
- : int list = [1; 2; 3; 4]
```

Exercice 80 : Liste d'éléments à tableau indicateur

Soit une constante entière $M \in \mathbb{N}$ fixée ($M = 5$ dans les exemples). On considère deux représentations en OCaml des sous-ensembles de $[\![0, M-1]\!]$ (par exemple l'ensemble $\{0, 2, 3\}$).

a) La première au moyen d'une liste finie des éléments de l'ensemble (`[0; 2; 3]` ou `[3; 2; 0]` pour l'exemple ci-dessus).

b) La seconde au moyen d'un tableau de taille M de booléen dont la case $i \in [\![0, M-1]\!]$ indique si oui ou non l'élément est dans l'ensemble (`[|true; false; true; true; false|]` pour l'exemple ci-dessus).

Q. 1 Donner des fonctions de conversion `a_to_b : int -> int list -> bool array` et `b_to_a : bool array -> int list` d'un type vers l'autre. La fonction `a_to_b` prend en paramètres non seulement la liste en question mais aussi l'entier M de l'énoncé.

Exercice 81 : Liste de tableaux

On considère une représentation des ensembles fini d'entiers au moyen d'une liste de tableaux de booléens. Une telle liste sera de la forme : `[`t_{p-1}`; `t_{p-2}`; ...; `t_1`; `t_0`]` où le tableau t_i est un tableau de 2^i booléen indiquant dans sa case $j \in [\![0, 2^{i-1}]\!]$ si oui ou non l'élément $2^i - 1 + j$ est, ou non dans l'ensemble représenté. Ainsi l'exemple `ex` ci-dessous représente l'ensemble d'entiers : $\{0, 2, 4, 5, 6, 8, 9, 14\}$.

```
let ex =
  (*  7      8      9      10     11     12     13     14 *)
  [[|false; true ; true ; false; false; false; false; true |];
   (* 3      4      5      6  *)
   [|false; true ; true ; true |];
   (* 1      2    *)
   [|false; true  |];
   (* 0   *)
   [|true |];
   ]
```

Q. 1 Définir une fonction `mem (i: int) (s: bool array list): bool` permettant de tester si un élément `i` est dans l'ensemble `s` représenté.

Q. 2 Définir une fonction `ajout (i: int) (s: bool array list): bool array list` permettant l'ajout d'un élément `i` à l'ensemble `s`. Par exemple l'appel `ajout 5 [[|true|]]` devra produire `[[|false; false; true; false|]; [|false; false|]; [|true|]]`.

Exercice 82 : Remplissage d'un tableau

Q. 1 Définir une fonction `remplissage : int array -> int list -> unit` prenant en paramètres un tableau d'entiers `t` et une liste *non vide* `l` et remplaçant, dans le tableau `t` les occurrences du nombre -1 par des valeurs piochées dans la liste `l`. Les valeurs de la liste `l` devront être utilisées dans l'ordre, si on atteint la fin de la liste on recommence. Par exemple si `t` est le tableau `[|1; 2; -1; 0; -2; -1; -1|]` l'appel `(remplissage t [4; 5; 6; 8])` l'appel devra modifier le tableau `t` en la valeur `[|1; 2; 4; 0; -2; 5; 6|]`. `(remplissage t [4; 5])` devra modifier le tableau `t` en la valeur `[|1; 2; 4; 0; -2; 5; 4|]`.

Exercice 83 : Fusion de listes en tableau

Q. 1 Définir une fonction `fusion : 'a list -> 'a list -> 'a array` prenant en paramètres deux listes *triées par ordre croissant* et retournant un tableau `t`, *trié* par ordre croissant et dont les éléments sont ceux se trouvant dans l'union des listes en paramètres.

Exercice 84 : Nombre d'occurrences

Q. 1 Définir une fonction `histogramme (donnees: int list) (pas: int) (max: int): int array` prenant en paramètres une liste d'entiers (`donnees`) dont les éléments sont tous dans l'intervalle entier $[\![0, \texttt{max} - 1]\!]$, un pas entier (`pas`) et une valeur entière `max` et retournant un tableau `t` de taille $\lceil \frac{\texttt{max}}{\texttt{step}} \rceil$ contenant dans sa case d'indice `i` un entier indiquant le nombre d'entiers de la liste `donnees` se trouvant dans l'intervalle entier $[\![i \times \texttt{pas}, (i+1) \times \texttt{pas} - 1]\!]$.

Exercice 85 : Sous-séquences monotones

Q. 1 Définir une fonction `sseq_monotones : 'a array -> int list` prenant en paramètres un tableau d'éléments et retournant la liste des longueurs des sous-séquences monotones du tableau en paramètre. La longueur d'une séquence monotone est le nombre d'élément se trouvant dans la séquence décrémenté de 1. Par exemple le tableau `[|1; 5; 8; 4; 6; 9; 8; 7; 6; 1; 9|]` contient les séquences monotones : `[|1; 5; 8|]`, `[|8; 4|]`, `[|4; 6; 9|]`, `[|9; 8; 7; 6; 1|]` et finalement `[|1; 9|]` de longueurs respectives : 2, 1, 2, 4, 1. Ainsi le résultat attendu pour cette entrée est la liste `[2; 1; 2; 4; 1]`.

Exercice 86 : Transposition

On considère des tableaux de taille n de listes d'entiers de $[\![0, n-1]\!]$. On appelle *transposé* d'un tel tableau t, un tableau t' de taille n, contenant dans sa case $i \in [\![0, n-1]\!]$ une liste telle que $\forall j \in [\![0, n-1]\!], j \in T[i] \Leftrightarrow i \in T[j]$.

Q. 1 Définir une fonction `transpose : int list array -> int list array` calculant la transposée du tableau qui les est passé en argument.

Exercice 87 : Matrice creuse

On considère dans cet exercice des matrices carrées d'entiers relatifs : $M \in \mathcal{M}_n(\mathbb{Z})$. Par exemple :
$\begin{pmatrix} 0 & 3 & 5 \\ 0 & 0 & 7 \\ -1 & 0 & 0 \end{pmatrix}$
On représente de telles matrices de deux manières :

- Ou bien par un tableau de tableau d'entiers relatifs, par exemple : `[|[|0; 3; 5|]; [|0; 0; 7|]; [|-1; 0; 0|]|]`.
- Ou bien par une liste de triplets : (i, j, v) indiquant que dans la case d'indice (i, j) de la matrice se trouve la valeur v. Si une case n'est pas mentionnée par une telle liste, c'est que la matrice représentée contient 0. Par exemple pour la matrice ci-dessus `[(0, 1, 3); (0, 2, 5); (1, 2, 7); (2, 0, -1)]`.

Q. 1 Donner des fonctions de conversion d'une représentation dans l'autre.

Exercice 88 : Cycle

Soit $n \in \mathbb{N}$. On dit d'une liste l d'entiers deux-à-deux distincts de $[\![0, n-1]\!]$ que c'est une liste *de cycle*. On dit qu'un tableau t' est le permuté d'un tableau t selon une liste de cycle $l = [l_0; l_1; \ldots; l_{p-1}]$ lorsque t' est obtenu à partir de t en déplaçant en case $l_{(i+1) \mod p}$ l'élément se trouvant en case l_i, et ce pour tout $i \in [\![1, p]\!]$.

Q. 1 Écrire une fonction `permute : int array -> int list -> int array` prenant en paramètres un tableau t de taille n et une liste de cycle l et calculant le permuté du tableau t selon la liste l.

Q. 2 Écrire une fonction `est_permute : int array -> int array -> int list` prenant en paramètres deux tableaux t et t' de taille n et retournant une liste de cycle l telle que t' est le permuté de t selon la liste l. Si une telle liste n'existe pas, on lèvera un message d'erreur.

Exercice 89 : Permutations

Dans cet exercice on considère uniquement des tableaux T de taille $n \geqslant 0$, contenant une et une seule fois chaque entier de $[\![0, n-1]\!]$. Étant donné un tableau et un entier $i \in [\![0, n-1]\!]$, on note $o_T(i)$ (on l'appelle l'ordre de i dans le tableau T) le plus petit entier $p > 0$ tel que $\underbrace{T[T[T[\ldots[T}_{p \text{ fois}}[i]]]]] = i$.

Q. 1 Définir une fonction `o : int array -> int -> int` prenant en paramètres un tableau T et un entier i et calculant $o_T(i)$.

Q. 2 Définir une fonction `max_o : int array -> int` prenant en paramètres un tableau T et retournant l'élément d'ordre maximal dans le tableau. On pourra supposer le tableau non vide.

Exercice 90 : Polynômes creux

On considère dans cet exercice des polynômes de $\mathbb{Z}[X]$. On représente de tels polynômes en OCaml au moyen d'une liste de couples d'entiers : la liste `[(`a_0`, `d_0`); (`a_1`, `d_1`); (`a_2`, `d_2`); …; (`a_p`, `d_p`)]` avec pour tout $i \in [\![0, p]\!]$, $a_i \in \mathbb{Z}^\star$ et $d_i \in \mathbb{N}^\star$. Une telle liste représente alors le polynôme :

$$X^{d_0}(a_0 + X^{d_1}(a_1 + X^{d_2}(a_2 + \ldots X^{d_{p-1}}(a_{p-1} + a_p X^{d_p}))))$$

On définit donc le type suivant.

```
type poly_1 = (int * int) list
```

On considère par ailleurs la représentation "classique" d'un polynôme $\sum_{i=0}^{n} b_i X^i$ au moyen d'un tableau de ses $n+1$ coefficients : `[|`b_0`; `b_1`; …; `b_n`|]`.
On définit donc le type suivant.

```
type poly_2 = int array
```

Q. 1 Donner des fonctions de conversion permettant la traduction d'une représentation vers l'autre. Discuter de la pertinence de l'une vis-à-vis de l'autre.

Exercice 91 : Énumérations

Dans cet exercice on s'intéresse au problème d'énumérer toutes les combinaisons possibles d'éléments choisis parmi des listes prédéfinies. Par exemple, étant donné les listes $l_0 =$ `[0; 1]`, $l_1 =$ `[1; 3; 5]` et $l_2 =$ `[9; 8]`, on souhaite pouvoir parcourir facilement l'ensemble des tableaux de taille 3 (le nombre de listes) dont la première case est choisie dans l_0, la seconde dans l_1 et la troisième dans l_2. Pour notre exemple cela donne les 12 tableaux de taille 3 suivants.

- `[|0; 1; 9|]`
- `[|1; 3; 8|]`
- `[|0; 5; 9|]`
- `[|1; 1; 8|]`
- `[|0; 3; 9|]`
- `[|1; 5; 8|]`
- `[|0; 1; 9|]`
- `[|1; 3; 8|]`
- `[|0; 5; 9|]`
- `[|1; 1; 8|]`
- `[|0; 3; 9|]`
- `[|1; 5; 8|]`

Étant donné p listes l_0, …, l_{p-1} d'entiers on souhaite donc énumérer tous les tableaux de taille p qu'il est possible de construire en plaçant dans ses cases d'indice i des éléments de l_i. On parlera d'*énumérations des tuples des* $(l_0, l_1, \ldots, l_{p-1})$. On appelle p la *dimension* de l'énumération.
On définit pour cela le type `enumeration` suivant :

```
type enumeration = (int list * int list) array
```

Ainsi une énumération est un tableau de couples de listes `[|`(k_0, l_0)`;` (k_1, l_1)`;` …`;` (k_{p-1}, l_{p-1})`|]`. La taille du tableau est la dimension de l'énumération.
Si l'on souhaite énumérer les tuples des $(l_0, l_1, \ldots, l_{p-1})$ on crée un tableau de taille p dont la case d'indice i contient initialement le couple (l_i, l_i). Durant l'énumération on ne modifie pas la seconde composante des couples (k_i, l_i) se trouvant dans les cases du tableau, ainsi celle-ci est invariante et vaut la liste l_i initiale.
L'*état* de l'énumération est obtenu en lisant les premiers entiers de chaque liste k_i du tableau d'énumération. Par exemple l'état de l'énumération `[|([0; 1], [0; 1]); ([3; 5], [1; 3; 5]); ([9; 8], [9; 8])|]` est `[|0; 3; 9|]`. On modifie une énumération pour qu'elle représente l'état suivant en :

- trouvant le plus petit entier i_0 tel que k_{i_0} n'est pas une liste de taille 1,
- elle est alors de la forme x_{i_0}`::`r_{i_0},
- on modifie alors :
  - k_{i_0} en r_{i_0},
  - toutes les listes k_j en l_j pour $j < i_0$.

Q. 1 Définir en OCaml les fonctions :

- `init : int list array -> enumeration` permettant la création d'une énumération ;
- `lit : enumeration -> int array` permettant la lecture de l'état d'une énumération ;
- `suivant : enumeration -> bool` permettant de passage d'un état de l'énumération au suivant.

La fonction suivant retournera un booléen indiquant si les itérations sont terminées.

```
# let e = init [| [0; 1] ; [1; 3; 5] ; [9; 8] |];;
val e : enumeration = [|([0; 1], [0; 1]); ([1; 3; 5], [1; 3; 5]); ([9; 8], [9; 8])|]
# lit e;;
- : int array = [|0; 1; 9|]
# suivant e;;
- : bool = true
# e;;
- : enumeration = [|([1], [0; 1]); ([1; 3; 5], [1; 3; 5]); ([9; 8], [9; 8])|]
# lit e;;
- : int array = [|1; 1; 9|]
# suivant e; suivant e; suivant e;;
- : bool = true
# e;;
- : enumeration = [|([0; 1], [0; 1]); ([5], [1; 3; 5]); ([9; 8], [9; 8])|]
# lit e;;
- : int array = [|0; 5; 9|]
# suivant e; suivant e; suivant e; suivant e; suivant e;;
- : bool = true
# lit e;;
- : int array = [|1; 3; 8|]
# suivant e; suivant e;;
- : bool = true
# lit e;;
- : int array = [|1; 5; 8|]
# suivant e;;
- : bool = false
```

Exercice 92 : Représentation d'arbres au moyen de tableaux

Soit $n \in \mathbb{N}$. Dans cet exercice, on s'intéresse à des arbres ayant n sommets, étiquetés par les entiers de l'intervalle $[\![0, n-1]\!]$ de sorte que chaque entier de $[\![0, n-1]\!]$ apparaît une et une seule fois. On représente un tel arbre en OCaml au moyen d'un tableau de taille n contenant des entiers de $[\![0, n-1]\!]$. Pour tout $i \in [\![0, n-1]\!]$:

- si `t.(i)` $= i$ alors i est la racine de l'arbre représenté,
- si `t.(i)` $\neq i$ alors le sommet étiqueté par `t.(i)` est le père du sommet i.

```
type arbre = int array
```

Q. 1 Dessiner l'arbre représenté par le tableau `[|1; 1; 1; 2; 3|]`.

Q. 2 Donner une fonction `racine : arbre -> int` prenant en paramètre un arbre représenté par tableau et retournant sa racine.

Q. 3 Donner une fonction `fabrique_branche : arbre -> int -> int list` prenant en paramètres un arbre et un sommet s et retournant la branche reliant s à la racine de l'arbre. Cette branche sera présentée sous la forme de la liste de ses sommets.

Q. 4 Donner une fonction `hauteur : arbre -> int` retournant la hauteur de l'arbre passé en paramètre.

Q. 5 Donner une fonction `plus_longue_branche : arbre -> int list` prenant en paramètre un arbre et retournant une branche de longueur maximale de l'arbre.

Exercice 93 : Programmation arbres 1

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction OCaml `sous_arbres : 'a btree -> 'a btree -> bool` permettant de tester si un arbre x est un sous-arbre d'un arbre y pour la relation d'ordre induite par la définition inductive des arbres binaires.

Exercice 94 : Programmation arbres 2

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Une *branche* dans un arbre binaire est un chemin entre la racine et une feuille de l'arbre. On définit la *valeur* d'une branche $(n_1, n_2, \ldots, n_p)$ comme la somme des étiquettes des nœuds des branches.

Q. 1 Définir une fonction OCaml `pg_branche: int btree -> int` calculant la valeur de la plus grande branche d'un arbre binaire étiqueté par des entiers.

Exercice 95 : Programmation arbres 3

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Étant donné un arbre étiqueté par des éléments de S, dont le parcours préfixe est $e_1, \ldots, e_n$, une valeur $x_0 \in A$, une fonction $f \in A \times S \to A$, on souhaite calculer la valeur :

$$f(\ldots f(f(x_0, e_1), e_2), \ldots, e_p)$$

Q. 1 Proposer une fonction OCaml calculant cette valeur.

Q. 2 En déduire une fonction calculant la taille d'un arbre binaire.

Q. 3 En déduire une fonction calculant le parcours préfixe d'un arbre binaire.

Exercice 96 : Programmation arbres 4

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

On définit une numérotation sur les noeuds d'un arbre de la manière suivante :
- La racine a le numéro 1 ;
- Si un noeud a numéro n, son fils gauche a pour noeud $2n$ et son fils droit pour numéro $2n+1$

Q. 1 Définir une fonction `numerote : 'a btree -> ('a * int) btree` ajoutant dans chaque noeud d'un arbre son numéro.

Exercice 97 : Programmation arbres 5

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction `est_parfait : 'a btree -> bool` permettant de tester qu'un arbre binaire est parfait.

Exercice 98 : Programmation arbres 6

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Définir une fonction `profondeur_min : 'a btree -> 'a -> int` telle que `(profondeur_min t a)` retournant la plus petite des profondeurs des noeuds d'étiquette `a` dans `t`. On lèvera un message d'erreur si une telle étiquette n'a pu être trouvée.

Exercice 99 : Programmation arbres 7

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Définir une fonction `encadre : int btree -> int -> int * int` prenant en arguments un arbre binaire de recherche b, une valeur x et retournant le plus petit encadrement de x par deux entiers se trouvant dans l'ensemble représenté par b. S'il n'est pas possible de majorer (resp. minorer) x on utilisera les valeurs spéciales `min_int` et `max_int`.

Exercice 100 : Programmation arbres 8

Q. 1 Proposer un algorithme permettant le calcul, à partir d'un arbre binaire b, d'un arbre binaire de recherche dont les étiquettes sont celles de b et dont la forme est celle de b. (Les étiquettes des noeuds ont été permutées).

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 2 Proposer une fonction OCaml implémentant cet algorithme.

Exercice 101 : Programmation arbres 9

On considère un arbre binaire de recherche dont les étiquettes sont deux à deux disjointes. On appelle chemin d'une étiquette e l'unique chemin menant au nœud d'étiquette e.

Q. 1 Proposer un algorithme permettant le calcul, du plus long chemin commun à deux étiquettes dans un arbre binaire de recherche.

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 2 L'implémenter.

Exercice 102 : Tranche d'un arbre binaire de recherche

```
type 'a btree =
  | V
  | N of 'a * 'a btree * 'a btree
```

Q. 1 Écrire une fonction `tranche : 'a btree -> 'a -> 'a -> 'a btree` qui reçoit comme paramètre un arbre binaire de recherche et deux valeurs et qui renvoie un arbre binaire de recherche dont les clés sont les clés de l'arbre qui sont comprises entre ces deux valeurs.

Exercice 103 : Successeur

On considère un arbre binaire de recherche dont les étiquettes sont deux à deux disjointes, représentant un ensemble S. Notons $\mathsf{succ}(x, S) = \min\{y \in S, y > x\}$ pour $x \in S$.

Q. 1 Donner un algorithme permettant le calcul de succ.

Q. 2 Proposer une implantation de succ en OCaml.

Exercice 104 : Nombre d'insertions

Étant donné une suite finie d'entiers $(u_i)_{i\in[\![1,n]\!]}$, on dénote par t_u l'arbre binaire de recherche obtenue par insertion successive de u_1, u_2, …, u_n dans l'arbre vide.

Q. 1 Exhiber deux suites distinctes u et v telles que $t_u = t_v$.

Q. 2 Étant donné un ensemble $\mathbb{P}$ de n noeuds non étiquetés ($\mathbb{P}$ donne donc la forme de l'arbre). Dénombrer (obtenir une relation de récurrence) le nombre de permutations σ de $\mathfrak{S}_n$ telles que $\mathsf{paths}(t_\sigma) = \mathbb{P}$.

Exercice 105 : Split

Q. 1 Définir une fonction `split : 'a btree -> 'a -> 'a btree * bool * 'a btree` prenant en arguments un arbre binaire de recherche t et une valeur x retournant :

- un arbre binaire de recherche g dont les étiquettes sont strictement inférieures à x
- un booléen indiquant si x est dans t.
- un arbre binaire de recherche d dont les étiquettes sont strictement supérieures à x

On devra assurer que l'ensemble des étiquettes de g augmenté des étiquettes de d est l'ensemble des étiquettes de t privé de x. La complexité de l'algorithme devra être en $\mathcal{O}h(t)$.

Q. 2 En déduire une fonction `union` calculant l'union de deux arbres binaires de recherche.