MP2I – DS1 2023-2024 – 4h

Pour chaque question, le langage imposé par l'énoncé doit être respecté.

Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.

Sauf précision contraire, les réponses doivent être justifiées.

Pour le code en C, on suppose que les fichiers d'en-tête `stdbool.h`, `stdint.h` et `stdio.h` sont inclus.

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

Correction :
Pour clarifier la correction, j'utiliserai *cette police pour mes commentaires, qui parfois seront des méta-corrections, parfois des réflexions qu'on peut être amené à faire au brouillon mais qui n'ont pas leur place sur une copie*, et la police habituelle pour ce qui doit vraiment apparaître dans une copie. (Bien sûr ce paragraphe déroge à la règle que je viens de poser.)
De façon générale, je souligne tout de suite qu'il ne suffit pas d'affirmer une chose pour qu'elle soit vraie, même avec le mot forcément *dans la phrase. On attend des preuves, c'est-à-dire des liens d'implication entre des hypothèses ou des propriétés déjà démontrées et de nouvelles propriétés.*

1 Questions de cours

Correction :
Aucune excuse pour ne pas savoir répondre à une question de cours. Pourtant je n'ai eu que 2 copies avec la bonne réponse aux trois questions, alors que j'en ai eu 10 qui ont 0 sur cette partie. Quitte à me répéter : sans le cours vous ne pouvez rien faire. Les plus malins réussiront à attraper quelques points sur les autres questions, mais de moins en moins au fur et à mesure de votre formation.

Question 1 : Où sont stockés les noms des fichiers dans un système de type unix/linux?

Correction :
Le nom d'un fichier est stocké dans le répertoire dans lequel il se trouve : un répertoire est une liste de couples (nom, numéro i-nœud).

Question 2 : Expliquer la différence entre un typage faible et un typage fort pour un langage de programmation.

Correction :
Le typage désigne la capacité d'un langage de programmation à faire des conversions de types implicites : plus le typage est faible, plus le langage peut faire de conversions implicites, plus il est fort, moins il le peut.

Question 3 : Le programme C suivant contient clairement une erreur :

```c
int main(void){
  int m = 0;
  m = 1 / m;
}
```

Cette erreur est-elle détectée au moment de la compilation ou est-elle détectée au moment de l'exécution ? (à justifier)

Correction :
L'erreur ici ne vient pas de l'absence de `return` *qui provoque uniquement un warning à la compilation. Ce n'était même pas un piège d'ailleurs, mais un oubli de ma part (on ne met pas systématiquement un* `return` *dans le* `main`*). Ce morceau de code avait été compilé, exécuté et expliqué devant vous en cours, il suffisait de redire ce que j'avais dit.*
L'erreur (une division par 0) est détectée à l'exécution car la compilation ne tient pas compte des valeurs des variables, mais uniquement de leurs types.

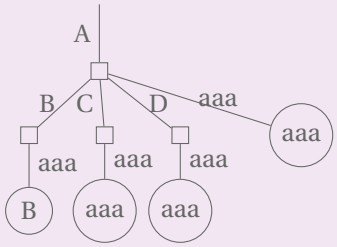
2 Exercices

Exercice 1 On exécute les commandes suivantes sur un shell :

```
$ cd
$ mkdir A
$ cd A
$ mkdir B
$ ls > B/aaa
$ mkdir C
$ ls B > aaa
$ cp aaa C
$ mkdir D
$ cp aaa D/aaa
```

Question 4 : Donner les contenus des fichiers `~/A/aaa`, `~/A/B/aaa`, `~/A/C/aaa` et `~/D/aaa` (aucune justification demandée).

Correction :
L'énoncé ne demande pas de justification, donc inutile d'en donner. Ça ne signifie pas qu'il ne faut pas réfléchir. Par ailleurs il ne sert à rien de répondre à une autre question, en disant par exemple "tous les fichiers ont le même contenu", ce qui ne répond pas à la question, ou en donnant la commande pour voir le contenu, ce qui ne répond toujours pas à la question.
À mon avis, une seule façon de trouver, c'est de créer à la main au brouillon l'arborescence :



Pour rappel, le symbole > redirige la sortie d'une commande vers un fichier.
contenu de `~/A/aaa` : `aaa`
contenu de `~/A/B/aaa` : `B`
contenu de `~/A/C/aaa` : `aaa`
contenu de `~/D/aaa` : ce fichier n'existe pas

Exercice 2 Écrire une fonction

```
unsigned int miroir(unsigned int n);
```

qui prend en argument un entier et renvoie l'entier obtenu en inversant l'ordre des chiffres en base 10.
Exemple : `miroir(123)` et `miroir(1230)` doivent renvoyer 321.

Correction :
Le but ici était de vous faire manipuler les chiffres de la représentation d'un entier en base 10. Il faut bien lire la spécification demandée et s'y tenir. L'énoncé ne demandait pas d'affichage, mais un retour.

```c
unsigned int miroir(unsigned int n){
  unsigned int res = 0;

  while(n > 0){
    res = 10 * res + n % 10;
    n = n / 10;
  }
  return res;
}
```

Exercice 3 On considère la fonction suivante :

```c
/** t, n : un tableau et son nombre de cases
 * renvoie la valeur maximale du tableau t */
int max(int *t, int n){
  int m = t[0];

  for(int i=1; i<n; i=i+1){
    if (t[i] > m){
      m = t[i];
    }
  }

  return m;
}
```

Question 5 : Prouver la correction totale de `max`.

Correction :
Il faut absolument faire une preuve rigoureuse, même si dans le cas présent la correction totale semble évidente. Soit je vous mets des exemples pour lesquels c'est dur de trouver variants et invariants parce que rien ne semble évident, et dans ce cas je risque d'avoir très peu de réponses, soit je mets des fonctions pour lesquelles c'est facile de comprendre ce qui se passe, mais vous devez montrer que vous savez le faire correctement. Cela vous préparera à traiter les exemples plus compliqués.
Montrer la correction totale, c'est montrer que l'exécution se termine pour toute entrée admissible et qu'elle donne la bonne réponse. À quel point rentrer dans les détails ? C'est la première question de ce type dans l'énoncé, il faut au moins montrer qu'on connaît les propriétés des objets manipulés. Quand une propriété est trivialement vraie, on la cite simplement, sans avoir à la justifier.

Montrons que l'exécution de `max` se termine pour toute entrée admissible.
Les instructions en dehors de la boucle `for` se terminent toutes.
La boucle `for` ne contient qu'un nombre fini d'itérations car $n-i$ en est un variant : c'est une expression entière *(ce point est évident à cause des types de `n` et `i`, ce n'est pas la peine de le justifier)*, positive (car `i<n`) et qui décroît strictement à chaque itération (`i=i+1`).

Attention : seuls deux d'entre vous savent que la condition précédente n'est pas suffisante pour prouver la sortie de boucle. Il faut aussi que chaque itération se termine.

Par ailleurs, chaque itération se termine car elle contient un test et éventuellement une affectation, donc l'exécution de la boucle `for` se termine, ce qui permet de conclure qu'il en est de même pour la fonction `max`.
Montrons la correction de la fonction `max`.
Montrons que le prédicat

(I) : `m` contient le maximum des cases 0 à `i-1` du tableau `t`

est un invariant pour la boucle `for`. Notons `t[0..k]` le sous-tableau de `t` formé des cases 0 à `k` comprises. *Il n'y a pas de notation en `C` pour extraire une tranche de tableau, il faut donc expliciter la notation employée.*

Vous pouvez constater que pour désigner l'itération, j'ai pris la notation de l'énoncé. Si j'en prend une autre, je dois expliquer ce qu'elle signifie, si je prends celle du code, c'est clair.
Une grande partie d'entre vous ne sait pas ce qu'est le maximum d'un ensemble d'entiers. Vous l'avez pourtant vu en math : c'est l'unique majorant qui appartient à cet ensemble. Il existe dès lors que l'ensemble est non vide (un

tableau `C` ne peut pas contenir 0 case, donc c'est vérifié) et fini (un tableau `C` ne peut pas contenir une infinité de cases, donc c'est vérifié). Il ne suffit donc pas de prouver qu'à la fin de la boucle `m` est un majorant.
(I) est vérifié avant la boucle : on a `m=t[0]` et `i=1`, `m` est donc bien le maximum de `t[0..0]`.
Supposons (I) vérifié au début d'une certaine itération `i` *(Certains semblent confondre boucle et itération, attention à utiliser le bon vocabulaire, sinon on ne comprend rien.)* et montrons qu'il l'est encore à la fin de l'itération.
Au début de l'itération `i`, la valeur de `m` est le maximum de `t[0..i-1]`.

- Si le maximum de `t[0..i]` se trouve dans `t[0..i-1]`, alors le test de la ligne 7 est faux et (I) est vérifié à la fin de l'itération `i`.
- Si le maximum de `t[0..i]` est `t[i]`, alors le test de la ligne 7 est vrai et `m` reçoit cette valeur à la ligne 8, et (I) est vérifié à la fin de l'itération `i`.

(I) est donc un invariant pour la boucle `for`. Comme on a prouvé qu'on sort de la boucle, (I) est vérifié en sortie de boucle, c'est-à-dire que la valeur de `m` renvoyée est le maximum de `t[0..n-1]`, c'est-à-dire le maximum de `t`.

Exercice 4 On considère la fonction suivante qui doit calculer le pgcd de deux entiers.

```
1  unsigned int pgcd(unsigned int a, unsigned int b){
2    if (a < b){
3      return pgcd(b, a);
4    }
5
6    if (b == 0){
7      return a;
8    }
9
10   return pgcd(a-b, b);
11 }
```

Question 6 : Prouver la correction totale de `pgcd`.

Correction :
Là encore, il s'agit de travailler votre rigueur avec des exemples pour lesquels les preuves sont plutôt simples. C'est le moyen de vous préparer à des exemples qui seront plus difficiles à prouver.
Un certain nombre d'entre vous parlent de boucles et d'itération. Ça n'a pas de sens ici, parce qu'il n'y a pas de boucle.
On a besoin d'un résultat de math pour la suite, montrons-le tout de suite. Soit deux entiers naturels non tous les deux nuls a et b tels que $a \geq b$. Notons $a \wedge b$ leur pgcd et montrons que $a \wedge b = (a-b) \wedge b$.

Soit k divisant a et b : il existe des entiers α et β tels que $a = \alpha k$ et $b = \beta k$, donc $a - b = (\alpha - \beta)k$ et k divise $a - b$, donc $a \wedge b$ divise $(a-b) \wedge b$.

Soit ℓ divisant $a - b$ et b : il existe des entiers γ et β tels que $a - b = \gamma\ell$ et $b = \beta\ell$, donc $a = (a-b) + b = (\gamma + \beta)\ell$ et ℓ divise a, donc $(a-b) \wedge b$ divise $a \wedge b$.

On en déduit que $a \wedge b = (a-b) \wedge b$.

Montrons maintenant la correction totale de la fonction `pgcd`, par récurrence sur la somme de ses paramètres qu'on considère au moins égale à 1 (se poser la question du pgcd de 0 et 0 n'a pas de sens).
Notez que je montre en même temps la terminaison et la correction. Pour cela il faut évidemment inclure les deux dans l'hypothèse de récurrence.
cas de base : Si $a + b = 1$, alors comme les deux paramètres sont des entiers non signés, on a nécessairement :

- soit $a = 1$ et $b = 0$, alors l'exécution se termine en ligne 7 et renvoie 1 qui est bien $1 \wedge 0$,
- soit $a = 0$ et $b = 1$, l'exécution aboutit alors à l'appel récursif `pgcd(1, 0)` de la ligne 3, qui est le cas précédent dont on a vu que l'exécution se termine et renvoie 1, qui est bien $0 \wedge 1$.

Notez que je commence par le cas direct, et ensuite je fais l'autre cas qui s'y ramène : comme j'ai déjà la réponse pour le cas direct, je peux m'en servir. Si je fais dans l'autre sens, je dois développer entièrement le cas qui a besoin d'une phase intermédiaire : c'est plus long et moins clair. Je ferai pareil ci-dessous.
hérédité : Supposons que pour un certain n, tout appel à `pgcd(a, b)` pour $a + b < n$ se termine et renvoie $a \wedge b$.
Montrons que c'est également le cas si $a + b = n$.

- si $a \geq b$:
  - si $b = 0$, alors l'appel se termine en ligne 7 et renvoie a qui vaut bien $a \wedge 0$,
  - si $b \neq 0$, alors l'appel récursif `pgcd(a-b,b)` de la ligne 10 est exécuté. Comme $(a-b) + b = a < a + b$, on sait par hypothèse de récurrence que cet appel se termine et renvoie $(a-b) \wedge b$ dont on a montré

qu'il est égal à $a \wedge b$;
- sinon : l'appel récursif `pgcd(b, a)` de la ligne 3 est exécuté ; on vient de voir que cet appel se termine et renvoie $b \wedge a$ qui est égal à $a \wedge b$.

Par le principe de la récurrence forte, on en déduit la correction totale de la fonction `pgcd`.

Exercice 5 On considère la fonction suivante qui implémente le crible d'Ératosthène :

```
1  /** premier : tableau de n booleens
2   * en sortie : premier[i] vaut vrai si et seulement si i est un nombre premier */
3  void eratosthene(bool *premier, int n){
4    int k;
5
6    premier[0] = false;
7    if (n>=2){
8      premier[1] = false;
9    }
10
11   for(k=2; k<n; k=k+1){
12     premier[k] = true;
13   }
14
15   k = 2;
16   while(k < n-1){
17     if(premier[k]){
18       for(int m=2; m*k<n; m=m+1){
19         premier[m*k] = false;
20       }
21     }
22     k = k+1;
23   }
24 }
```

Question 7 : Prouver la correction totale de la fonction `eratosthene`.

Correction :
Cette fonction contient 3 boucles, il faut donc bien spécifier de laquelle on parle à chaque fois, sinon c'est nébuleux. C'est l'occasion d'utiliser le fait qu'on a déjà montré qu'on connaissait la définition d'un variant et d'un invariant.
Le but de la boucle `for` de la ligne 12 est de remplir les cases 2 à $n-1$ du tableau `premier` avec des `true`. L'exécution de cette boucle se termine (toutes les itérations se terminent et elle possède un variant : $n-k$) et est correcte (invariant : les cases 2 à $k-1$ contiennent la valeur `true`). *Vous notez que j'ai spécifié le but de la boucle : il n'est pas dans le code de l'énoncé, si je me contente de dire que la boucle est correcte, je n'ai rien dit. Un morceau de code ne peut être correct que vis-à-vis de sa spécification.*
Il reste deux boucles à traiter : la boucle `while` et la boucle `for` de la ligne 18. Comme la seconde est imbriquée dans la première, on a besoin de savoir qu'elle se termine pour dire que chaque itération de la première se termine, donc il faut absolument commencer par la boucle interne. Comme dit en cours d'ailleurs.
Le but de la boucle `for` de la ligne 18 est de mettre à `false` toutes les cases du tableau `premier` dont l'indice est un multiple de k, sans modifier les valeurs des autres cases *(là encore, je spécifie le but de la boucle avant de dire qu'elle fait son travail)*. Cette boucle se termine (chaque itération se termine et elle possède un variant : $n-mk$) et est correcte (invariant : les cases dont l'indice est un multiple de k entre $2k$ et $(m-1)k$ compris ont la valeur `false`, les autres ont leurs valeurs initiales).
La boucle `while` se termine : on vient de montrer que chacune de ses itérations se termine, et $n-k$ est un variant pour cette boucle, assurant un nombre fini d'itérations.
Montrons qu'à la fin de la boucle `while`, une case de `premier` contient `true` si et seulement si son indice est un nombre premier.
Il est clair que `premier[0]` et `premier[1]` valent `false` : cette valeur est assignée en ligne 8 et ne change plus. Les entiers 0 et 1 ne sont pas premiers, donc c'est correct.
Montrons que le prédicat

(I) : les cases de `premier` d'indice au moins 2 contiennent `false` si et seulement si leur indice possède un diviseur propre dans $[\![2, k-1]\!]$.

(I) est vérifié avant la boucle car si $k=2$, alors l'ensemble $[\![2,k-1]\!]$ est vide et on a vu qu'à l'issue de la boucle `for` de la ligne 11, toutes les cases de `premier` d'indice au moins 2 contiennent `true`.
Supposons (I) vrai au début de l'itératon k et montrons qu'il est encore vrai à la fin de cette itération.
On a vu que la boucle `for` de la ligne 18 ne modifie pas les cases dont les indices ne sont pas des multiples de k *(d'où l'intérêt d'avoir traité la boucle imbriquée avant)*. Si une telle case est à `true`, son indice possède un diviseur propre dans $[\![2,k-1]\!]$ d'après (I), si elle est à `false`, son indice ne possède pas de diviseur propre dans $[\![2,k-1]\!]$, comme son indice n'est pas un multiple de k, il ne possède pas de diviseur propre dans $[\![2,k]\!]$ et la valeur de la case est encore à `false` à la fin de l'itération.
Si l'indice d'une case est un multiple de k, alors on a vu que l'action de la boucle `for` de la ligne 18 est d'affecter `true` à la valeur de cette case.
À la fin de l'itération k, on a donc : les cases de `premier` d'indice au moins 2 contiennent `false` si et seulement si leur indice possède un diviseur propre dans $[\![2,k]\!]$. On en déduit que (I) est un invariant pour la boucle `while`.
Comme on sait qu'on sort de la boucle `while`, on en déduit que (I) est vérifié en sortie de la boucle, soit : les cases de `premier` d'indice au moins 2 contiennent `false` si et seulement si leur indice possède un diviseur propre dans $[\![2,n-2]\!]$.
Comme aucun indice de case de `premier` ne peut posséder $n-1$ comme diviseur propre, on a le résultat souhaité.

Exercice 6 On considère l'algorithme de tri suivant :

Algorithme 1 Tri par sélection

entrée : un tableau T de n entiers
sortie : le tableau T est trié dans l'ordre croissant
1 : **si** $n=1$ **alors** l'algorithme se termine
2 : **fin si**
3 : on parcourt le tableau T pour trouver son élément maximal, soit m,
4 : on inverse cet élément avec celui situé à la dernière case du tableau T,
5 : on trie récursivement les $n-1$ premières cases de T.

Question 8 : Prouver que cet algorithme possède une correction totale.

Correction :
Encore une fois, quand il n'y a pas de boucle (ce qui est le cas ici), on ne parle pas de boucle, d'itération, de variant de boucle ou d'invariant, ça n'a pas de sens.
L'algorithme de tri par sélection se termine car l'expression n est un variant d'appel (expression entière positive dont la valeur diminue strictement à chaque appel récursif) *(c'est la première utilisation de cet objet dans la copie, on rappelle la définition ; c'est évident que l'objet vérifie la définition ici)* et toutes les instructions en dehors des appels récursifs se terminent.
Montrons par récurrence sur la valeur du paramètre $n \geq 1$ que l'algorithme est correct.
cas de base : Si $n=1$, l'algorithme se termine en ligne 1, sans avoir modifié le contenu de T qui était trié puisque ne contenant qu'une valeur.
hérédité : Supposons que l'algorithme soit correct pour toute entrée de taille n, pour un certain entier $n \geq 1$, et montrons qu'il est correct pour toute entrée de taille $n+1$.
Soit un tableau T de taille $n+1$ qu'on donne en entrée à l'algorithme de tri par sélection. Comme $n \geq 1$, alors $n+1>1$, donc l'appel ne sort pas en ligne 1.
Après l'exécution de la ligne 4, le plus grand élément de T est dans sa dernière case, c'est-à-dire à l'endroit où il doit être pour que les éléments se trouvent dans l'ordre croissant. L'appel récursif de la ligne 5 sur les n premières cases de T ordonne dans l'ordre croissant les n premiers éléments de T par hypothèse de récurrence. T est donc trié dans l'ordre croissant après la ligne 5.
Attention : tous ceux qui ont essayé de le faire par récurrence l'ont fait à l'envers, en entrant dans les appels récursifs jusqu'au sommet de la pile d'appel. Ce raisonnement est fait à l'envers : il faudrait a minima expliquer ensuite qu'on ne modifie pas l'ordre en revenant dans la fonction appelante. Plutôt que de raisonner sur le nombre d'appels récursifs restants, c'est beaucoup plus simple ici (et toujours en fait) de raisonner sur la taille des données.

Question 9 : Traduire cet algorithme en fonction `C` de signature

```
1 void tri_selection(int *T, int n);
```

Correction :

```c
/** T : tableau de longueur n
 * renvoie l'indice du max de T
 */
int indice_max(int *T, int n){
  int m = 0;

  for(int i=1; i<n; i=i+1){
    if(T[i] > T[m]){
      m = i;
    }
  }
  return m;
}

void tri_selection(int *T, int n){
  int m, tmp;

  if(n==1){
    return;
  }
  m = indice_max(T, n);
  tmp = T[n-1];
  T[n-1] = T[m];
  T[m] = tmp;
  tri_selection(T, n-1);
}
```

3 Problème 1 : Suite de Fibonacci et décomposition de Zackendorf

Suite de Fibonacci

On rappelle la définition de la suite de Fibonacci :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geq 2 \end{cases}$$

Question 10 : Écrire une fonction qui prend en argument un entier n et renvoie la valeur de F_n.

Correction :

Je ne demandais rien d'intelligent, ni de rapide (on n'est pas sur machine et on n'a pas vu de technique particulière pour accélerer les choses). Le code qui suit est à proscrire en pratique, mais il me suffisait largement dans le cadre de ce DS.

```c
int fibonacci(int n){
  if(n == 0 || n == 1){
    return n;
  }
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Décomposition de Zeckendorf

Pour $n \in \mathbb{N}$, on appelle *décomposition de Zeckendorf de n* une décomposition de n comme somme de termes distincts d'indices supérieurs ou égal à 2 de la suite de Fibonacci, de telle manière qu'il n'y ait pas deux termes d'indices consécutifs dans la somme. Autrement dit, il existe des indices $c_1, c_2, \ldots, c_k$ tels que :

- $c_0 \geq 2$,
- pour tout $i < k$, on a $c_{i+1} > c_i + 1$ (pas d'indices consécutifs),

- $n = \sum_{i=0}^{k} F_{c_i}$.

Par exemple :

- $F_3 + F_5 = 2 + 5 = 7$ est une décomposition de Zeckendorf de 7 (avec $c_0 = 3$ et $c_1 = 5$),
- $F_4 + F_5 = 3 + 5 = 8$ **n'est pas** une décomposition de Zeckendorf de 8 car F_4 et F_5 sont deux termes consécutifs de la suite de Fibonacci,
- $F_6 = 8$ est une décomposition de Zeckendorf de 8 (avec $c_0 = 6$).

Question 11 : Déterminer une décomposition de Zeckendorf de 20, 21 et 22.

Correction :
Pas grand chose à justifier ici, c'est du calcul élémentaire. Vous pouvez vous contenter de mettre les premières valeurs de la suite de Fibonacci, puis les décompositions demandées.

n	0	1	2	3	4	5	6	7	8
F_n	0	1	1	2	3	5	8	13	21

Donc :

$$\begin{aligned} 20 &= F_3 + F_5 + F_7 \\ 21 &= F_8 \\ 22 &= F_2 + F_8 \end{aligned}$$

Question 12 : Démontrer par récurrence que tout entier strictement positif admet une décomposition de Zeckendorf (on admet qu'elle est unique).

Correction :
Peu d'entre vous se sont rendus compte de la subtilité de cette question. Pourtant les exemples de la question précédente auraient pu vous mettre la puce à l'oreille. De façon générale, un problème est une suite de questions qui ont un lien entre elles : il n'est pas rare que le résultat d'une question puisse être réutilisé dans une question qui suit, soit directement, soit pour suggérer quelque chose. Ici, vous pouvez sans doute remarquer que la décomposition de 21 n'est pas égale à la décomposition de 20 plus F_2 : cette technique est donc sans doute trop réductrice. Par ailleurs, vous pouvez aussi vous interrogez sur la façon dont vous avez construit les décompositions pour ces exemples : il est probable que vous ayez pris le plus grand entier de la suite de Fibonacci et que vous ayez ensuite cherché à le compléter, sans tomber sur une contradiction. C'est donc la direction à explorer en premier.
Beaucoup se sont lancés dans des preuves par récurrence, ce qui est une bonne idée en soit, mais n'ont pas su poser la propriété à démontrer, voire n'ont pas compris que le k et les c_i de la formule dépendaient de n dans la définition de la décomposition, écrivant alors des choses absurdes.
Montrons que tout entier $n \in \mathbb{N}^*$ possède une décomposition de Zeckendorf.
cas de base : c'est vérifié pour $n = 1$ puisque $1 = F_2$
hérédité :Supposons que pour un certain $n \in \mathbb{N}^*$, tous les entiers de $[\![1, n]\!]$ possèdent une décomposition de Zeckendorf et montrons que c'est également le cas de $n+1$.
Si $n+1 = F_k$ pour un certain entier $k \geq 2$, on a immédiatement la décomposition. Sinon il existe un entier $k \geq 2$ tel que $F_k < n+1 < F_{k+1}$ car la suite de Fibonacci croît et tend vers l'infini. Considérons l'entier $m = n+1-F_k$.
On a $m < n+1$ car $F_k > 0$, donc m possède une décomposition de Zeckendorf par hypothèse de récurrence.
De plus $m < F_{k+1} - F_k$ par choix de k, donc $m < F_{k-1}$ par définition de la suite de Fibonacci. On peut en déduire que le plus grand indice qui apparaît dans une décomposition de Zeckendorf de m est au plus $k-2$. Donc si on ajoute F_k à une décomposition de Zeckendorf de m, on obtient une décomposition de Zeckendorf de $n+1$.
Par le principe de récurrence forte, tout entier strictement positif possède une décomposition de Zeckendorf.

Codage de Fibonacci

On appelle *codage de Fibonacci* d'un entier positif k un tableau `tab` de 0 et de 1 indiquant par des 1 les indices des termes de la suite de Fibonacci utilisés dans la décomposition de Zeckendorf de k (`lst[0]` indique alors si F_2 est utilisé dans la représentation). On remarque que par définition de la décomposition de Zeckendorf, deux cases consécutives de `lst` ne peuvent contenir toutes les deux la valeur 1.

0	1	0	1

est ainsi un codage de Fibonacci de 7.

Question 13 : Écrire une fonction `decode` qui prend en argument un tableau d'entiers supposé représenter le codage de Fibonacci d'un entier n (sans avoir à faire de vérifications) et sa taille, et renvoie la valeur de n.

Correction :
Encore une fois, on ne demandait pas quelque chose d'intelligent, autant aller au plus simple pour l'instant.

```
1 int decode(int *t, int lg){
2   int n = 0;
3
4   for(int i=0; i<lg; i=i+1){
5     n = n + t[i] * fibonacci(i+2);
6   }
7   return n;
8 }
```

4 Problème 2 : Représentation binaire gauche des entiers naturels

4.1 Mise en place

Soient m un entier naturel et N un entier naturel non nul. Il est classique d'appeler *représentation binaire standard de l'entier m sur N chiffres*, ou plus simplement *représentation standard*, toute suite finie $b = (b_n)_{0\leq n<N}$ de longueur N telle que, pour tout indice n compris entre 0 et $N-1$, le chiffre b_n appartient à $\{0,1\}$ et l'égalité suivante est vérifiée

$$m = \sum_{n=0}^{N-1} b_n 2^n.$$

Définition : Nous appelons *représentation binaire gauche de l'entier m sur N chiffres* toute suite finie $g = (g_n)_{0\leq n<N}$ de longueur N telle que,

(i) pour tout indice n compris entre 0 et $N-1$, le chiffre g_n appartient à $\{0,1,2\}$,

(ii) l'égalité suivante est satisfaite

$$m = \sum_{n=0}^{N-1} g_n(2^{n+1}-1),$$

(iii) le chiffre « 2 » n'apparaît qu'au plus une fois parmi les chiffres $(g_n)_{0\leq n<N}$,

(iv) s'il existe une position p telle que le chiffre g_p est 2, alors, pour tout indice n compris entre 0 et $p-1$, le chiffre g_n est nul.

De manière plus courte, nous parlons simplement de *représentation gauche*.
La figure 1 ci-dessous donne une représentation standard et une représentation gauche sur quatre chiffres des seize premiers entiers. Conformément à l'usage habituel, nous écrivons toute représentation, qu'elle soit standard ou gauche, sous la forme d'un mot $b_{N-1}\cdots b_0$ ou $g_{N-1}\cdots g_0$ dans lequel les chiffres de poids faibles sont écrits à droite.

Entier	Repr. standard	Repr. gauche
0	0000	0000
1	0001	0001
2	0010	0002
3	0011	0010
4	0100	0011
5	0101	0012
6	0110	0020
7	0111	0100

Entier	Repr. standard	Repr. gauche
8	1000	0101
9	1001	0102
10	1010	0110
11	1011	0111
12	1100	0112
13	1101	0120
14	1110	0200
15	1111	1000

FIGURE 1 – Représentations standard et gauche des seize premiers entiers

Question 14 : Soit c un entier. Donner la représentation standard de l'entier dont une représentation gauche est $10\cdots0$ (avec c chiffres nuls) en justifiant sommairement. Faire de même avec l'entier dont une représentation gauche est $20\cdots0$ (avec c chiffres nuls).

Correction :
Le tableau de la figure 1 permet de se faire une intuition de la réponse, et c'est très bien d'avoir l'idée de passer par là. Mais il ne constitue en rien une preuve du résultat. Ici il faut faire des calculs pour arriver au résultat. L'entier dont la représentation gauche est $1\underbrace{0\cdots0}_{c}$ est

$$1\times(2^{c+2}-1)=\sum_{i=0}^{c+1}2^i,$$

donc sa représentation standard est

$$\underbrace{11\ldots1}_{c+1}.$$

L'entier dont la représentation gauche est $2\underbrace{0\cdots0}_{c}$ est $2\times(2^{c+2}-1)=2a$, où a est l'entier précédent, donc sa représentation standard est

$$\underbrace{11\ldots1}_{c+1}0.$$

Question 15 : Montrer que le plus grand entier naturel qui admet une représentation gauche sur N chiffres est $M_N=2^{N+1}-2$. Préciser la représentation gauche de cet entier.

Correction :
Presque tous ceux qui ont traité cette question ont commencé par dire que le plus grand entier représentable par représentation gauche sur N chiffres est celui dont la représentation gauche est $20\ldots0$. Ce résultat est à prouver, sinon c'est juste une affirmation en l'air.
Certains ont essayé en disant que le plus grand entier représentable a forcément un chiffre de poids fort à 2. Mais c'est pareil : il faut le prouver. En base 3 c'est clair parce que dès lors qu'on augmente un chiffre, on augmente l'entier représenté, mais ici la condition (iv) ne nous autorise pas à considérer l'augmentation d'un chiffre sans contexte.
Montrons par récurrence sur $N\geq 0$ que le plus grand entier obtenu par représentation gauche sur N chiffres est M_N, et que sa représentation gauche est $2\underbrace{0\cdots0}_{N-1}$.
cas de base : La figure 1 montre que cette propriété est vraie pour N allant de 0 à 3.
hérédité : Supposons que pour un certain N, le plus grand entier représentable à gauche sur N chiffres soit M_N et que sa représentation gauche est $2\underbrace{0\cdots0}_{N-1}$.
Le plus grand entier représentable sur $N+1$ chiffres est alors d'une des formes suivantes *(je fais une disjonction de cas suivant le chiffre de poids fort)* :

- $2\underbrace{0\cdots0}_{N}$ qui vaut $2(2^{N+1}-1)=2^{N+2}-2=M_{N+1}$,
- $1g_{N-1}\ldots g_0$ qui vaut la somme de $2^{N+1}-1$ et de la valeur maximale dont la représentation gauche est $g_{N-1}\ldots g_0$; par hypothèse de récurrence, cette valeur maximale est $M_N=2^{N+1}-2$, le nombre représenté est donc majoré par $2^{N+1}-1+2^{N+1}-2=2^{N+2}-3<M_{N+1}$,
- $g_{N-1}\ldots g_0$ qui vaut la valeur maximale dont la représentation gauche est $g_{N-1}\ldots g_0$; par hypothèse de récurrence, cette valeur maximale est $M_N=2^{N+1}-2$, le nombre représenté est donc majoré par $2^{N+1}-2<M_{N+1}$,

On en déduit le résultat souhaité.

Définition : Soit n_0 un indice compris entre 0 et $N-1$. On dit que l'indice n_0 est la *position du chiffre de plus fort poids* d'une représentation $g_{N-1}\cdots g_0$ si l'indice n_0 est le plus petit entier tel que, pour tout indice $n>n_0$, le chiffre g_n est nul. On appelle le chiffre g_{n_0} le *chiffre de plus fort poids*.

Question 16 : Soient N un entier naturel non nul, $g=g_{N-1}\cdots g_0$ et $h=h_{N-1}\cdots h_0$ deux représentations gauches d'un même entier m. Démontrer que les chiffres de plus fort poids de g et de h sont de même valeur et à la même position.

Correction :
Plusieurs ont tenté ici d'invoquer l'unicité de la représentation gauche. Mais c'est justement l'objet de cette question et de la question suivante, on ne peut donc pas supposer qu'on sait que c'est vrai.
Notons n_g la position du chiffre de plus fort poids de g et n_h celle de h. En particulier, on a : $g_{n_g} \in \{1,2\}$.
Montrons par l'absurde que les chiffres de poids fort de g et h ont la même position.
Supposons, sans perte de généralité, que $n_g > n_h$. En particulier, on a alors : $h_n = 0$ pour tout entier $n \in [\![n_h+1, n_g]\!]$ (ensemble qui contient au moins n_g).
Comme m est représenté à gauche par $g_{N-1} \dots g_0$ et par $h_{N-1} \dots h_0$, on en déduit que :

$$m = \sum_{n=0}^{n_g} g_n(2^{n+1}-1) = \sum_{n=0}^{n_h} h_n(2^{n+1}-1).$$

Comme $n_g > n_h$, on en déduit que :

$$M_{n_g} < 2^{n_g+1}-1 \leq g_{n_g}(2^{n_g+1}-1) = \sum_{n=0}^{n_g-1}(h_n-g_n)(2^{n+1}-1) \leq \underbrace{\sum_{n=0}^{n_g-1} h_n(2^{n+1}-1)}_{A}.$$

Le nombre A est clairement représentable à gauche sur n_g chiffres (c'est l'écriture de la formule), ce qui est contradictoire avec le fait qu'il est strictement supérieur à M_{n_g} qui d'après la question précédente est le plus grand entier représentable à gauche sur n_g chiffres.
Donc les chiffres de poids forts des deux représentations g et h sont en même position.
Montrons par l'absurde que les chiffres de poids fort de g et h ont la même valeur.
Supposons, sans perte de généralité, que $g_{n_g} > h_{n_h}$. En particulier : $g_{n_g} - h_{n_h} \in \{1,2\}$
Comme précédemment, on a :

$$m = \sum_{n=0}^{n_g} g_n(2^{n+1}-1) = \sum_{n=0}^{n_g} h_n(2^{n+1}-1).$$

Et on en déduit :

$$M_{n_g} < (g_{n_g}-h_{n_g})(2^{n_g+1}-1) = \sum_{n=0}^{n_g-1}(h_n-g_n)(2^{n+1}-1) \leq \underbrace{\sum_{n=0}^{n_g-1} h_n(2^{n+1}-1)}_{A}$$

qui permet d'aboutir à la même contradiction pour A.
On conclut que les chiffres de poids fort de g et h ont même valeur.

Question 17 : Démontrer par récurrence sur $N \geq 0$ que tout entier appartenant à l'intervalle $[\![0, M_N]\!]$, où l'entier M_N a été introduit à la question 15, ne possède au plus qu'une seule représentation gauche sur N chiffres.

Correction :
Montrons qu'un entier de $[\![0, M_N]\!]$ possède au plus une représentation gauche sur N chiffres, par récurrence sur $N \geq 0$:
cas de base : $M_0 = 0$ et 0 ne possède qu'une seule représentation gauche sur 0 chiffres.
hérédité Supposons que le résultat soit vrai pour tout entier inférieur ou égal à un certain N et montrons qu'il est vrai pour $N+1$.
Soit $g = g_N \cdots g_0$ et $h = h_N \cdots h_0$ deux représentations gauches d'un même entier $m \in [\![0, M_{N+1}]\!]$. D'après la question précédente, on sait que les chiffres de poids fort de g et h se situent à la même position (notons-la n) et ont même valeur. On en déduit que $g' = g_{n-1} \dots g_0$ et $h' = h_{n-1} \dots h_0$ représentent à gauche le même entier, sur n chiffres. Comme $n < N+1$, on peut appliquer l'hypothèse de récurrence et déduire que g' et h' sont identiques.
Par le principe de récurrence, tout entier de $[\![0, M_N]\!]$ possède au plus une représentation gauche sur N chiffres.
Comme $[\![0, M_N]\!]$ est de cardinal $2^{N+1}-1$ et qu'il existe le nombre de représentations gauches sur N chiffres est

$$\sum_{k=0}^{N-1} \underbrace{2^{N-(k+1)}}_{2 \text{ à l'indice } k} + \underbrace{2^N}_{\text{pas de 2}} = \sum_{k=0}^{N} 2^k = 2^{N+1}-1,$$

on en déduit que chaque entier de $[\![0, M_N]\!]$ possède exactement une représentation gauche sur N chiffres.

Dans la suite on suppose que $N = 8$ et on s'intéresse aux représentations gauches des entiers de $[\![0, 2^9 - 2]\!]$.
On encode en C une telle représentation gauche par un tableau d'entiers de N cases où la i-ème case contient le i-ème chiffre de cette représentation ($0 \leq i < 8$).

Question 18 : Écrire le corps de la fonction suivante :

```c
/** g : tableau de 8 cases
 * renvoie vrai si r est une représenation gauche d'un entier, false sinon */
bool est_rg(int *g);
```

Correction :
Quand on écrit du code sur feuille, il faut privilégier la clarté, encore plus que sur machine, parce que le correcteur ne peut pas exécuter en cas de doute. Donc il faut commenter!
Certains ont confondu le fait que la représentation gauche s'écrit avec le chiffre de poids faible qui est numéroté 0 à droite, ce qui est une convention, et le fait que par convention on représente les tableaux dans l'autre sens. Il faut bien entendu que le chiffre de poids faible se trouve dans la case 0 du tableau, sinon tout devient illisible et compliqué car il faut jongler avec les indices.

```c
bool est_rg(int *g){
  for(int i=0; i<8; i=i+1){ // condition (i)
    if(g[i]<0 || g[i]>2){
      return false;
    }
  }

  bool deja_vu = false;
  int ind = -1;
  for(int i=0; i<8; i=i+1){ // condition (iii)
    if(g[i] == 2){
      if(deja_vu){
        return false;
      }
      deja_vu = true;
      ind = i; // pour la condition suivante
    }
  }

  for(int i=0; i<ind; i=i+1){ // condition (iv)
    if(g[i] != 0){
      return false;
    }
  }

  return true;
}
```

Question 19 : Écrire une fonction C

```c
int rg_to_int(int *g)
```

qui renvoie -1 si g n'est la représentation gauche d'aucun entier, ou l'entier dont g est la représentation gauche sinon.

Correction :
*J'ai déjà dit et rappelé maintes fois qu'il n'y a pas d'opérateur puissance en C. Arrêtez donc d'écrire des ^ ou des ^^ ou encore des **, ça ne fonctionne pas.*

```c
int rg_to_int(int *g){
  if(!est_rg(g)){
    return -1;
```

```
4     }
5
6     int m = 0;
7     int p = 1; // pour obtenir la bonne puissance de 2
8     for(int n=0; n<8; n=n+1){
9       p = p*2;
10      m = m + g[n]*(p-1);
11    }
12
13    return m;
14  }
```

4.2 Incrémentation et décrémentation

Nous proposons l'algorithme suivant :

Algorithme 2 Algorithme mystère

entrée : représentation gauche $g = g_{N-1} \cdots g_0$ d'un certain entier $m < M_N$
sortie : modification de g
1 : **si** aucun des chiffres $(g_n)_{0 \leq n < N}$ ne vaut 2 **alors**
2 : changer le chiffre g_0 en $g_0 + 1$
3 : **sinon** en notant p la position du chiffre 2, changer le chiffre g_p en 0 et le chiffre g_{p+1} en $g_{p+1} + 1$
4 : **fin si**

Question 20 : Vérifier que la valeur de g après l'application de l'algorithme mystère est encore une représentation gauche d'un entier.
Notons m' l'entier dont la représentation gauche est g après exécution de l'algorithme mystère. Caractériser, en fonction de l'entier m, la valeur de l'entier m' (justifier).

Correction :
Il faut ici montrer que la valeur finale de g vérifie encore les propriétés (I), (iii) et (iv). Il ne suffit pas de le dire, il faut fournir des arguments, comme toujours. Je présente par propriété, on peut aussi séparer en fonction de ce qui s'exécute dans le programme, ça n'a pas d'importance.
La condition (i) est vérifiée à la fin de l'algorithme puisque les seules modifications éventuellement faites sont les suivantes :

- ligne 2 : g_0 est incrémenté de 1 ; comme on est dans le cas où sa valeur d'origine est 0 ou 1, sa nouvelle valeur est donc 1 ou 2 ;
- ligne 3 : on change g_p en 0, ce qui ne contredit pas (i), et on incrémente g_{p+1} ; comme on est dans le cas où la valeur initiale de g_p est 2, on sait que la valeur initiale de g_{p+1} est 0 ou 1 (condition (iii)), sa nouvelle valeur est donc 1 ou 2.

La condition (iii) est vérifiée à la fin de l'algorithme puisque :

- si aucun des chiffres initiaux de g n'est 2, on ne modifie la valeur que d'un chiffre (lignes 2 et 3), donc on ne peut pas passer plus d'un chiffre à 2 ;
- sinon on passe l'unique chiffre qui était à 2 à 0 et on modifie un seul autre chiffre, donc on ne peut pas avoir plus d'un chiffre à 2 à la fin.

La condition (iv) est vérifiée à la fin de l'algorithme puisque :

- si aucun des chiffres initiaux de g n'est 2, le seul chiffre qui peut valoir 2 à la fin est g_0, qui n'est suivi par aucun chiffre dans la représentation ;
- sinon le seul chiffre qui peut valoir 2 à la fin est g_{p+1}, or les g_k pour $k \in [\![0, p-1]\!]$ sont nuls au début de l'algorithme (condition (iv)) et leur valeur n'est pas modifiée et g_p vaut 0 à la fin de l'algorithme.

Il ne suffit pas de décrire l'algorithme avec des mots pour en déduire que la nouvelle valeur représentée est $m+1$: il faut faire des calculs. Je détaille beaucoup ici pour ceux qui ont du mal à calculer, mais bien sûr on peut sauter des étapes.
Notons g', g'_n et m' respectivement la représentation, le n-ème chiffre de la représentation et l'entier représenté à la fin de l'algorithme.

On a

$$m = \sum_{n=0}^{N-1} g_n(2^{n+1}-1) \quad \text{et} \quad m' = \sum_{n=0}^{N-1} g'_n(2^{n+1}-1).$$

Si l'algorithme est passé par la ligne 2, on a :

$$\begin{aligned} m' &= \sum_{n=0}^{N-1} g'_n(2^{n+1}-1) \\ &= \left(\sum_{n=1}^{N-1} g_n(2^{n+1}-1)\right) + g_0 + 1 \\ &= \left(\sum_{n=1}^{N-1} g_n(2^{n+1}-1)\right) + g_0(2^{0+1}-1) + 1 \\ &= \left(\sum_{n=0}^{N-1} g_n(2^{n+1}-1)\right) + 1 \\ &= m+1 \end{aligned}$$

Si l'algorithme est passé par la ligne 3, on a :

$$\begin{aligned} m' &= \sum_{n=0}^{N-1} g'_n(2^{n+1}-1) \\ &= \sum_{n=p+1}^{N-1} g'_n(2^{n+1}-1) \\ &= \left(\sum_{n=p+2}^{N-1} g_n(2^{n+1}-1)\right) + (g_{p+1}+1)(2^{p+2}-1) \\ &= \left(\sum_{n=p+1}^{N-1} g_n(2^{n+1}-1)\right) + 2^{p+2} - 1 \\ &= \left(\sum_{n=p+1}^{N-1} g_n(2^{n+1}-1)\right) + 2^{p+2} - 2 + 1 \\ &= \left(\sum_{n=p+1}^{N-1} g_n(2^{n+1}-1)\right) + 2(2^{p+1}-1) + 1 \\ &= \left(\sum_{n=p+1}^{N-1} g_n(2^{n+1}-1)\right) + g_p(2^{p+1}-1) + 1 \\ &= \left(\sum_{n=p}^{N-1} g_n(2^{n+1}-1)\right) + 1 \\ &= \left(\sum_{n=0}^{N-1} g_n(2^{n+1}-1)\right) + 1 \\ &= m+1 \end{aligned}$$

Question 21 : Écrire une fonction `c`

```
1 /** g : tableau de 8 cases encodant une représentation gauche d'un entier m
2  * retour : vrai si on peut modifier g pour que l'entier représenté soit m+1,
3  * faux sinon
4  * si c'est possible, g représente l'entier m+1, sinon son contenu n'est pas modifié
5  */
6 bool rg_incr(int *g);
```

Correction :
Il suffit de suivre l'algorithme. Si on a bien pris soin de sortir quand on n'a rien à incrémenter, alors on est sûr que

s'il y a un 2 il est unique, et autant parcourir la représentation du chiffre de poids fort au chiffre de poids faible : si on croise un 2, on applique la ligne 3 de l'algorithme, sinon on applique la ligne 2. Attention à bien vérifier qu'on ne dépasse pas des limites du tableau, par exemple en évitant d'écrire `g[i+1]` si `i` peut prendre la valeur 7.

```
1  bool rg_incr(int *g){
2    if(!est_rg(g) || g[7] == 2){ // pas une représentation gauche
3      return false;                // ou le plus grand d'après la question 15
4    }
5    for(int i=6; i>=0; i=i-1){ // g[7] != 2 sinon on est sorti
6      if(g[i] == 2){
7        g[i] = 0;
8        g[i+1] = g[i+1] + 1;
9        return true;
10     }
11   }
12   // pas de 2
13   g[0] = g[0] + 1;
14   return true;
15 }
```

Question 22 : Écrire une fonction `c`

```
1 /** g : tableau de 8 cases encodant une représentation gauche d'un entier m
2  * retour : vrai si on peut modifier g pour que l'entier représenté soit m-1,
3  * faux sinon
4  * si c'est possible, g représente l'entier m-1, sinon son contenu n'est pas modifié
5  */
6 bool rg_decr(int *g);
```

Correction :

Ici il faut analyser ce qui se passe pour écrire la fonction puisque l'algorithme n'est pas décrit et n'est pas complètement trivial.

Si g est bien une représentation gauche, en supposant qu'il représente un entier m non nul, alors :

- soit le chiffre de poids faible g_0 est non nul, et g est obtenu à partir de la représentation gauche $m-1$ en passant par la ligne 2 de l'algorithme d'incrémentation, donc il faut décrémenter g_0,
- soit le chiffre de poids faible g_0 est nul, et g est obtenu à partir obtenu à partir de la représentation gauche $m-1$ (soit g') en passant par la ligne 3 de l'algorithme d'incrémentation, il faut donc trouver le premier chiffre non nul de g et on sait que c'est g'_{p+1}.

```
1  bool rg_decr(int *g){
2    if(!est_rg(g)){
3      return false;
4    }
5    if(g[0] != 0){
6      g[0] = g[0] - 1;
7      return true;
8    }
9    for(int i=1; i<8; i++){
10     if(g[i] != 0){
11       g[i-1] = 2;
12       g[i] = g[i] - 1;
13       return true;
14     }
15   }
16   return false;
17 }
```