

## Table des matières

<b>1 Exercice 1 : Le Calendrier Perpétuel</b>	<b>1</b>
1.1 Analyse du problème . . . . .	1
1.2 Entrées et sorties . . . . .	1
1.2.1 Entrées . . . . .	1
1.2.2 Sorties . . . . .	1
1.3 Implémentation et explication . . . . .	1
1.3.1 Structures de données . . . . .	1
1.3.2 Algorithme principal : <code>jour_dans_semaine</code> . . . . .	2
1.4 Complexité algorithmique . . . . .	2
1.4.1 Fonction <code>Isbissextile(year)</code> . . . . .	2
1.4.2 Fonction <code>nbr_jour_dans_mois(month, year)</code> . . . . .	2
1.4.3 Fonction <code>jour_dans_semaine(day, month, year)</code> . . . . .	3
1.5 Tests et résultats . . . . .	3
1.6 Code complet . . . . .	3
<b>2 Exercice 2 : Suite de Fibonacci</b>	<b>6</b>
2.1 Analyse du problème . . . . .	6
2.2 Entrées et sorties . . . . .	6
2.2.1 Entrées . . . . .	6
2.2.2 Sorties . . . . .	6
2.3 Implémentation et explication . . . . .	6
2.3.1 Algorithme récursif naïf . . . . .	6
2.3.2 Algorithme itératif . . . . .	7
2.4 Complexité algorithmique . . . . .	7
2.4.1 Version récursive naïve . . . . .	7
2.4.2 Version itérative . . . . .	7
2.5 Tests et résultats . . . . .	8
2.6 Code complet . . . . .	8
<b>3 Exercice 3 : Calcul de Puissance</b>	<b>10</b>
3.1 Analyse du problème . . . . .	10
3.2 Entrées et sorties . . . . .	10
3.3 Complexité algorithmique . . . . .	10
3.4 Implémentation et explication . . . . .	10
3.5 Code complet . . . . .	10
3.6 Tests et résultats . . . . .	10

## 1 Exercice 1 : Le Calendrier Perpétuel

### 1.1 Analyse du problème

Le problème consiste à créer un programme capable de déterminer le jour de la semaine correspondant à une date donnée (jour, mois, année). L'algorithme doit gérer les particularités du calendrier grégorien pour garantir l'exactitude du résultat.

Les points clés à considérer sont :

- **Années bissextilles** : Une année est bissextile si elle est divisible par 4, à l'exception des années séculaires (multiples de 100) qui ne sont pas divisibles par 400. Par exemple, 2000 est bissextile, mais 1900 ne l'est pas.
- **Nombre de jours par mois** : Le nombre de jours varie (28, 29, 30, ou 31) et dépend du mois et de l'année (pour février).
- **Date de référence** : L'algorithme se base sur une date de référence connue. Dans notre cas, le 1er janvier 1900, qui était un lundi.

L'approche consiste à calculer le nombre total de jours écoulés entre la date de référence et la date cible, puis à utiliser une opération modulo 7 pour trouver le jour de la semaine.

## 1.2 Entrées et sorties

### 1.2.1 Entrées

- `day` : entier représentant le jour du mois (1-31).
- `month` : énumération `Month` représentant le mois (de JANUARY à DECEMBER).
- `year` : entier représentant l'année (ex : 2025).

### 1.2.2 Sorties

- Une énumération de type `Day` représentant le jour de la semaine (de MONDAY à SUNDAY).

## 1.3 Implémentation et explication

### 1.3.1 Structures de données

Pour améliorer la lisibilité et la robustesse du code, nous utilisons des énumérations pour les jours et les mois.

```
1 typedef enum Day { MONDAY, TUESDAY, ..., SUNDAY } Day;
2 typedef enum Month { JANUARY, FEBRUARY, ..., DECEMBER } Month;
```

Listing 1 – Énumérations pour les jours et les mois

### 1.3.2 Algorithme principal : jour\_dans\_semaine

L'algorithme se déroule en quatre étapes :

1. **Initialisation** : Un compteur de jours `days` est initialisé à 0.
2. **Calcul des jours des années complètes** : Une boucle parcourt chaque année de 1900 jusqu'à l'année précédant celle de la date cible. Pour chaque année, on ajoute 366 jours si elle est bissextile, et 365 sinon.
3. **Calcul des jours des mois complets** : Une seconde boucle parcourt les mois de l'année en cours, de janvier jusqu'au mois précédent celui de la date cible. Pour chaque mois, on ajoute le nombre de jours correspondant.
4. **Ajout des jours du mois courant** : On ajoute le numéro du jour dans le mois (moins 1, car le premier jour est déjà compté).
5. **Calcul final** : Le total `days` est divisé par 7, et le reste donne le jour de la semaine (0 pour lundi, 1 for mardi, etc.).

```

1 Day jour_dans_semaine(int day, Month month, int year)
2 {
3     int days = 0;
4
5     // 1. Compter les jours des années complètes
6     for (int i = 1900; i < year; i++) {
7         days += Isbissextile(i) ? 366 : 365;
8     }
9
10    // 2. Ajouter les jours des mois complets
11    for (int i = 0; i < month; i++) {
12        days += nbr_jour_dans_mois(i, year);
13    }
14
15    // 3. Ajouter les jours du mois en cours
16    days += day - 1;
17
18    // 4. Le jour est le reste modulo 7
19    return days % 7;
20 }
```

Listing 2 – Algorithme de calcul du jour de la semaine

## 1.4 Complexité algorithmique

### 1.4.1 Fonction Isbissextile(year)

- **Complexité temporelle** :  $O(1)$ , car elle effectue un nombre constant d'opérations arithmétiques.
- **Complexité spatiale** :  $O(1)$ , car elle n'utilise pas de mémoire supplémentaire proportionnelle à l'entrée.

### 1.4.2 Fonction nbr\_jour\_dans\_mois(month, year)

- **Complexité temporelle** :  $O(1)$ , car la structure **switch** a un nombre de cas fixe.
- **Complexité spatiale** :  $O(1)$ .

### 1.4.3 Fonction jour\_dans\_semaine(day, month, year)

- **Complexité temporelle** :  $O(Y + M)$ , où  $Y$  est le nombre d'années depuis 1900 et  $M$  est le nombre de mois dans l'année. Comme  $M$  est au maximum 12, la complexité est dominée par  $Y$ , soit  $O(Y)$ .
- **Complexité spatiale** :  $O(1)$ .

## 1.5 Tests et résultats

L'algorithme a été validé avec un ensemble de dates pour vérifier sa correction, y compris :

- 7 décembre 2025 → SUNDAY (dimanche)
- 11 février 2024 → dimanche

— 1er janvier 1900 → MONDAY (lundi)  
 Les résultats obtenus correspondent aux jours de la semaine attendus, confirmant la validité de l'implémentation.

## 1.6 Code complet

```

1 #ifndef CALENDAR_H
2 #define CALENDAR_H
3
4 #include <stdbool.h>
5
6 typedef enum Day
7 {
8     MONDAY,
9     TUESDAY,
10    WEDNESDAY,
11    THURSDAY,
12    FRIDAY,
13    SATURDAY,
14    SUNDAY,
15 } Day;
16
17 typedef enum Month
18 {
19     JANUARY,
20     FEBRUARY,
21     MARCH,
22     APRIL,
23     MAY,
24     JUNE,
25     JULY,
26     AUGUST,
27     SEPTEMBER,
28     OCTOBER,
29     NOVEMBER,
30     DECEMBER
31 } Month;
32
33 bool Isbissextile(int year);
34 int nbr_jour_dans_mois(Month month, int year);
35 Day jour_dans_semaine(int day, Month month, int year);
36
37 #endif

```

Listing 3 – calendar.h - Fichier d'en-tête

```

1 #include "calendar.h"
2
3 bool Isbissextile(int year)
4 {
5     return year % 4 == 0 && (year % 100 != 0 || year % 400 ==
0);

```

```

6  }
7
8 int nbr_jour_dans_mois(Month month, int year)
9 {
10    bool bissextie = Isbissextile(year);
11    switch (month)
12    {
13        case JANUARY:
14        case MARCH:
15        case MAY:
16        case JULY:
17        case AUGUST:
18        case OCTOBER:
19        case DECEMBER:
20            return 31;
21
22        case APRIL:
23        case JUNE:
24        case SEPTEMBER:
25        case NOVEMBER:
26            return 30;
27
28        case FEBRUARY:
29            return bissextie ? 29 : 28;
30
31    default:
32        return 0;
33    }
34 }
35
36 Day jour_dans_semaine(int day, Month month, int year)
37 {
38    int days = 0;
39    for (int i = 1900; i < year; i++)
40    {
41        days += Isbissextile(i) ? 366 : 365;
42    }
43
44    for (int i = 0; i < month; i++)
45    {
46        days += nbr_jour_dans_mois(i, year);
47    }
48    days += day - 1;
49
50    return days % 7;
51 }
```

Listing 4 – calendar.c - Implémentation complète

## 2 Exercice 2 : Suite de Fibonacci

### 2.1 Analyse du problème

La suite de Fibonacci est une séquence d'entiers où chaque terme est la somme des deux termes qui le précèdent. Elle est définie par la relation de récurrence suivante :

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n - 1) + F(n - 2) \text{ pour } n > 1 \end{aligned}$$

L'objectif est d'implémenter deux algorithmes pour calculer le  $n$ -ième terme de la suite : une version récursive directe et une version itérative, afin de comparer leurs performances.

### 2.2 Entrées et sorties

#### 2.2.1 Entrées

- $n$  : entier positif, l'indice du terme de la suite à calculer.

#### 2.2.2 Sorties

- Le  $n$ -ième terme de la suite de Fibonacci. Le type de retour est `unsigned long long` pour la version itérative afin de gérer de grands nombres, tandis que la version récursive utilise `int` pour illustrer les limites de débordement.

### 2.3 Implémentation et explication

#### 2.3.1 Algorithme récursif naïf

Cet algorithme traduit directement la définition mathématique de la suite.

- **Cas de base (conditions d'arrêt)** :
  - Si  $n = 0$ , la fonction retourne 0.
  - Si  $n = 1$ , la fonction retourne 1.
- **Étape récursive** : Pour tout autre  $n$ , la fonction s'appelle elle-même avec  $n - 1$  et  $n - 2$  et retourne la somme de leurs résultats.

Bien qu'élégant, cet algorithme est très inefficace car il recalcule les mêmes termes de la suite de nombreuses fois.

```

1 int fibonacci_rec(int n)
2 {
3     if (n <= 1) {
4         return n;
5     }
6     return fibonacci_rec(n - 1) + fibonacci_rec(n - 2);
7 }
```

Listing 5 – Algorithme récursif de Fibonacci

### 2.3.2 Algorithme itératif

Cet algorithme calcule les termes de la suite de manière séquentielle en utilisant une boucle, ce qui évite les calculs redondants.

1. **Initialisation** : On initialise deux variables pour stocker les deux premiers termes, `pprev = 0` ( $F(0)$ ) et `prev = 1` ( $F(1)$ ).
2. **Itération** : Une boucle `while` s'exécute de 2 à  $n$ . À chaque itération, on calcule le terme suivant en additionnant `pprev` et `prev`.
3. **Mise à jour** : Les valeurs de `pprev` et `prev` sont mises à jour pour l'itération suivante.
4. **Résultat** : Après la boucle, la variable contenant le dernier terme calculé est retournée.

```

1  unsigned long long fibonacci_iter(int n)
2  {
3      if (n <= 1) {
4          return n;
5      }
6      unsigned long long pprev = 0;
7      unsigned long long prev = 1;
8      unsigned long long fibo = 0;
9      for (int i = 2; i <= n; i++) {
10         fibo = pprev + prev;
11         pprev = prev;
12         prev = fibo;
13     }
14     return prev;
15 }
```

Listing 6 – Algorithme itératif de Fibonacci

## 2.4 Complexité algorithmique

### 2.4.1 Version récursive naïve

- **Complexité temporelle** :  $O(2^n)$ . L'arbre d'appels récursifs a une profondeur de  $n$  et chaque appel (sauf ceux des cas de base) en génère deux autres, menant à une croissance exponentielle.
- **Complexité spatiale** :  $O(n)$ , en raison de la profondeur de la pile d'appels récursifs.

### 2.4.2 Version itérative

- **Complexité temporelle** :  $O(n)$ . L'algorithme effectue une seule boucle qui s'exécute  $n - 1$  fois.
- **Complexité spatiale** :  $O(1)$ , car il utilise un nombre constant de variables, quelle que soit la valeur de  $n$ .

## 2.5 Tests et résultats

Les performances des deux algorithmes ont été mesurées en utilisant la fonction `clock()` pour différentes valeurs de  $n$ .

<b>n</b>	<b>Itératif (<math>\mu s</math>)</b>	<b>Récuratif (<math>\mu s</math>)</b>
10	< 1	< 1
20	< 1	13
30	< 1	1,213
40	< 1	173,778
45	< 1	1,662,728

TABLE 1 – Comparaison des temps d'exécution

### Observations :

- Pour de petites valeurs de  $n$ , les deux algorithmes sont rapides.
- Dès que  $n$  dépasse 30, le temps d'exécution de la version récursive augmente de manière exponentielle, devenant rapidement impraticable.
- La version itérative reste quasi instantanée, démontrant sa supériorité en termes d'efficacité.
- **Cas de débordement (edge case)** : Pour  $n > 46$ , le résultat de la version récursive (utilisant `int`) déborde et devient incorrect. La version itérative avec `unsigned long long` peut calculer des termes beaucoup plus grands (jusqu'à  $n \approx 93$ ).

## 2.6 Code complet

```

1
2
3 int fibonacci_rec(int n)
4 {
5
6     if (n == 0)
7     {
8         return 0;
9     }
10    if (n == 1)
11    {
12        return 1;
13    }
14
15    return fibonacci_rec(n - 1) + fibonacci_rec(n - 2);
16 }
17
18 unsigned long long fibonacci_iter(int n)
19 {
20
21     if (n <= 1)
22     {
23         return n;
24     }

```

```
25
26     int count = 2;
27
28     unsigned long long prev = 1;
29     unsigned long long pprev = 0;
30
31     int fibo = 0;
32     while (count <= n)
33     {
34         fibo = pprev + prev;
35
36         pprev = prev;
37         prev = fibo;
38
39         count++;
40     }
41     return fibo;
42 }
```

Listing 7 – fibonacci.c - Implémentation complète

### 3 Exercice 3 : Calcul de Puissance

#### 3.1 Analyse du problème

*À compléter*

#### 3.2 Entrées et sorties

*À compléter*

#### 3.3 Complexité algorithmique

*À compléter*

#### 3.4 Implémentation et explication

*À compléter*

#### 3.5 Code complet

*À compléter*

#### 3.6 Tests et résultats

*À compléter*