Table des matières

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 bissextiles** : 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 précédant 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.).

```
Day jour_dans_semaine(int day, Month month, int year)
{
    int days = 0;

    // 1. Compter les jours des annees completes
    for (int i = 1900; i < year; i++) {
        days += Isbisextile(i) ? 366 : 365;
    }

    // 2. Ajouter les jours des mois complets
    for (int i = 0; i < month; i++) {
        days += nbr_jour_dans_mois(i, year);
    }

    // 3. Ajouter les jours du mois en cours
    days += day - 1;

    // 4. Le jour est le reste modulo 7
    return days % 7;
}
```

Listing 2 – Algorithme de calcul du jour de la semaine

1.4 Complexité algorithmique

1.4.1 Fonction `Isbisextile(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 → dimanchep
— 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

```c
#ifndef CALENDAR_H
#define CALENDAR_H

#include <stdbool.h>

typedef enum Day
{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY,
} Day;

typedef enum Month
{
    JANUARY,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
} Month;

bool Isbisextile(int year);
int nbr_jour_dans_mois(Month month, int year);
Day jour_dans_semaine(int day, Month month, int year);

#endif
```

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

```cpp
#include "calendar.h"

bool Isbisextile(int year)
{
    return year % 4 == 0 && (year % 100 != 0 || year % 400 ==
       0);
}

int nbr_jour_dans_mois(Month month, int year)
{
    bool bissextie = Isbisextile(year);
    switch (month)
    {
    case JANUARY:
    case MARCH:
    case MAY:
    case JULY:
    case AUGUST:
    case OCTOBER:
    case DECEMBER:
        return 31;

    case APRIL:
    case JUNE:
    case SEPTEMBER:
    case NOVEMBER:
        return 30;

    case FEBRUARY:
        return bissextie ? 29 : 28;

    default:
        return 0;
    }
}

Day jour_dans_semaine(int day, Month month, int year)
{
    int days = 0;
    for (int i = 1900; i < year; i++)
    {
        days += Isbisextile(i) ? 366 : 365;
    }

    for (int i = 0; i < month; i++)
    {
        days += nbr_jour_dans_mois(i, year);
    }
    days += day - 1;

    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 :

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

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.

```
unsigned long long fibonacci_iter(int n)
{
    if (n <= 1) {
        return n;
    }
    unsigned long long pprev = 0;
    unsigned long long prev = 1;
    unsigned long long fibo = 0;
    for (int i = 2; i <= n; i++) {
        fibo = pprev + prev;
        pprev = prev;
        prev = fibo;
    }
    return prev;
}
```

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.

n	Itératif (μs)	Récursif (μs)
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

Le problème consiste à calculer x^n (la puissance d'un nombre) de manière efficace. Bien que l'opération semble simple, il existe plusieurs approches avec des performances très différentes. L'objectif est d'implémenter et de comparer trois algorithmes :

— Une approche récursive naïve
— Une approche itérative naïve
— Une approche récursive optimisée utilisant l'exponentiation binaire

Les points clés à considérer sont :

— **Cas de base** : $x^0 = 1$ pour tout x
— **Propriété de l'exponentiation binaire** :
  — Si n est pair : $x^n = (x^{n/2})^2$
  — Si n est impair : $x^n = x \times (x^{(n-1)/2})^2$
— **Gestion des grands nombres** : Utilisation du type `unsigned long long` pour éviter les débordements

3.2 Entrées et sorties

3.2.1 Entrées

— `base` : entier représentant la base x
— `exposant` : entier positif représentant l'exposant n

3.2.2 Sorties

— Le résultat de x^n de type `unsigned long long` (capacité jusqu'à 18,446,744,073,709,551,615)

3.3 Implémentation et explication

3.3.1 Algorithme récursif naïf

Cet algorithme traduit directement la définition mathématique : $x^n = x \times x^{n-1}$

— **Cas de base** : Si $n = 0$, retourner 1
— **Étape récursive** : Retourner $x \times puissance(x, n-1)$

```
ull puissance_recursif_naif(int base, int exposant) {
    if (exposant == 0) {
        return 1;
    }
    return base * puissance_recursif_naif(base, exposant - 1);
}
```

Listing 8 – Algorithme récursif naïf

3.3.2 Algorithme itératif naïf

Cet algorithme utilise une boucle simple pour multiplier la base par elle-même n fois.

1. **Initialisation** : `resultat = 1`

2. **Itération** : Boucle de 0 à $n-1$, à chaque itération : `resultat = resultat * base`
3. **Résultat** : Retourner `resultat`

```
ull puissance_iteratif_naif(int base, int exposant) {
    if (exposant == 0) return 1;

    ull resultat = 1;
    for (int i = 0; i < exposant; i++) {
        resultat = resultat * base;
    }
    return resultat;
}
```

Listing 9 – Algorithme itératif naïf

3.3.3 Algorithme récursif optimisé (Exponentiation binaire)

Cet algorithme exploite la propriété mathématique de l'exponentiation pour réduire le nombre de multiplications nécessaires.

1. **Cas de base** :
   — Si $n = 0$, retourner 1
   — Si $n = 1$, retourner x
2. **Calcul récursif** : Calculer `temp = puissance(x, n/2)` une seule fois
3. **Cas pair** : Si n est pair, retourner `temp * temp`
4. **Cas impair** : Si n est impair, retourner `x * temp * temp`

```
ull puissance_recursif_optimise(int base, int exposant) {
    if (exposant == 0) return 1;
    if (exposant == 1) return base;

    ull temp = puissance_recursif_optimise(base, exposant / 2);

    if (exposant % 2 == 0) {
        return temp * temp;
    } else {
        return base * temp * temp;
    }
}
```

Listing 10 – Algorithme récursif optimisé (exponentiation binaire)

3.4 Complexité algorithmique

3.4.1 Version récursive naïve

— **Complexité temporelle** : $O(n)$. L'algorithme effectue exactement n appels récursifs.
— **Complexité spatiale** : $O(n)$, en raison de la profondeur de la pile d'appels récursifs.

3.4.2 Version itérative naïve

— **Complexité temporelle** : $O(n)$. La boucle s'exécute exactement n fois.
— **Complexité spatiale** : $O(1)$, car seules quelques variables locales sont utilisées.

3.4.3 Version récursive optimisée (exponentiation binaire)

— **Complexité temporelle** : $O(\log n)$. À chaque appel récursif, l'exposant est divisé par 2, ce qui donne une profondeur d'arbre récursif de $\log_2 n$.
— **Complexité spatiale** : $O(\log n)$, proportionnelle à la profondeur de la pile d'appels.

3.5 Tests et résultats

Le programme de test permet de mesurer les performances des trois algorithmes en calculant x^n pour des valeurs données par l'utilisateur.

Exemple de test : Calcul de 2^{30}

Algorithme	Résultat	Temps
Récursif Naïf	1,073,741,824	$\sim$ 0.000015 sec
Itératif Naïf	1,073,741,824	$\sim$ 0.000010 sec
Récursif Optimisé	1,073,741,824	$\sim$ 0.000005 sec

TABLE 2 – Comparaison des performances pour 2^{30}

Observations :

— Pour des exposants modérés (jusqu'à 30-40), les trois algorithmes sont rapides car les processeurs modernes effectuent les multiplications très rapidement.
— L'algorithme récursif optimisé (exponentiation binaire) effectue le moins d'opérations : environ $\log_2 n$ multiplications contre n multiplications pour les versions naïves.
— Pour des exposants très grands (supérieurs à 1000), la différence de performance devient significative, l'algorithme optimisé étant nettement plus rapide.
— **Avantage de l'exponentiation binaire** : Pour $n = 1000$, la version naïve effectue 1000 multiplications, tandis que la version optimisée n'en effectue qu'environ 10 ($\log_2 1000 \approx 10$).
— La version itérative naïve a une meilleure localité mémoire que la version récursive naïve, ce qui peut la rendre légèrement plus rapide en pratique.

3.6 Code complet

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

/* Definition du type pour simplifier et gerer les grands
   nombres */
/* unsigned long long permet d'aller jusqu'a
   18,446,744,073,709,551,615 */
typedef unsigned long long ull;
```

```
8
9  /* 1. Fonction Puissance Recursive Naive */
10 ull puissance_recursif_naif(int base, int exposant) {
11     /* Cas de base */
12     if (exposant == 0) {
13         return 1;
14     }
15     /* Appel recursif : x * x^(n-1) */
16     return base * puissance_recursif_naif(base, exposant - 1);
17 }
18
19 /* 2. Fonction Puissance Iterative Naive (O(n)) */
20 ull puissance_iteratif_naif(int base, int exposant) {
21     if (exposant == 0) return 1;
22
23     ull resultat = 1;
24     /* Boucle simple qui multiplie 'base' par elle-meme
          'exposant' fois */
25     for (int i = 0; i < exposant; i++) {
26         resultat = resultat * base;
27     }
28     return resultat;
29 }
30
31 /* 3. Fonction Puissance Recursive Optimisee (Exponentiation
      Binaire) (O(log n)) */
32 ull puissance_recursif_optimise(int base, int exposant) {
33
34     if (exposant == 0) return 1;
35     if (exposant == 1) return base;
36
37     /* Appel recursif : on calcule la moitie de la puissance
          UNE SEULE FOIS */
38     ull temp = puissance_recursif_optimise(base, exposant / 2);
39
40     if (exposant % 2 == 0) {
41         /* Si pair : (x^(n/2))^2 */
42         return temp * temp;
43     } else {
44         /* Si impair : x * (x^((n-1)/2))^2 */
45         /* Note : en C, exposant/2 fait deja la division
              entiere (n-1)/2 */
46         return base * temp * temp;
47     }
48 }
49
50 int main() {
51     int base, exposant;
52     clock_t debut, fin;
53     double temps_ecoule;
54     ull resultat;
```

```c

    printf("=== TEST DE PERFORMANCE DES ALGORITHMES DE PUISSANCE ===\n");
    printf("Entrez la base (entier) : ");
    /* (void) permet d'ignorer le retour de scanf sans warning */
    (void)scanf("%d", &base);

    printf("Entrez l'exposant (entier positif) : ");
    (void)scanf("%d", &exposant);

    printf("\nCalcul de %d^%d :\n", base, exposant);
    printf("-------------------------------------------------------\n");

    /* --- Test 1 : Recursif Naif --- */
    debut = clock();
    resultat = puissance_recursif_naif(base, exposant);
    fin = clock();
    temps_ecoule = ((double)(fin - debut)) / CLOCKS_PER_SEC;
    printf("[Recursif Naif]    Resultat: %llu | Temps: %f sec\n", resultat, temps_ecoule);

    /* --- Test 2 : Iteratif Naif --- */
    debut = clock();
    resultat = puissance_iteratif_naif(base, exposant);
    fin = clock();
    temps_ecoule = ((double)(fin - debut)) / CLOCKS_PER_SEC;
    printf("[Iteratif Naif]    Resultat: %llu | Temps: %f sec\n", resultat, temps_ecoule);

    /* --- Test 3 : Recursif Optimise --- */
    debut = clock();
    resultat = puissance_recursif_optimise(base, exposant);
    fin = clock();
    temps_ecoule = ((double)(fin - debut)) / CLOCKS_PER_SEC;
    printf("[Recursif Opti]    Resultat: %llu | Temps: %f sec\n", resultat, temps_ecoule);

    printf("-------------------------------------------------------\n");

    return 0;
}
```

Listing 11 – puissance.c - Implémentation complète