U.S.T.H.B - 2024-2025
Faculté d'Informatique
N. Bensaou & M.N Djouada

Master Informatique Visuelle
Algorithmique et Complexité (M1 - MIV)

TP01 : La complexité pratique d'un programme

Exercice 1.- La fonction `clock()`.

La fonction `clock()` en C fait partie de la bibliothèque standard `<time.h>` et est utilisée pour mesurer le temps CPU écoulé depuis le début de l'exécution du programme. Elle retourne le nombre de "ticks" d'horloge (unités de temps) écoulés.

Prototype

```
#include <time.h>
clock_t clock(void);
```

Détails

- La fonction renvoie une valeur de type `clock_t`, qui représente le nombre de cycles d'horloge consommés par le programme depuis son démarrage.
- Si la fonction échoue, elle renvoie la constante `-1`.
- La macro `CLOCKS_PER_SEC` (définie dans `<time.h>`) représente le nombre de "ticks" par seconde. Cela permet de convertir le résultat de `clock()` en secondes.

Exemple d'utilisation

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

int main() {
        clock_t start, end;
        double cpu_time_used;

        // Capture le temps de depart
        start = clock();

        // Section de code dont on veut mesurer le temps d'execution
        for (int i = 0; i < 1000000; i++);

        // Capture le temps de fin
        end = clock();

        // Calcul du temps ecoule en secondes
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

        printf("Temps CPU  : %lf secondes\n", cpu_time_used);

        return 0;
}
```

Explication

— **Capture du temps de départ** : `start = clock();`
— **Exécution d'une section de code** : Une simple boucle `for` est exécutée.
— **Capture du temps de fin** : `end = clock();`
— **Calcul du temps écoulé** : La différence entre `end` et `start` donne le nombre de ticks écoulés, qui est ensuite converti en secondes en divisant par `CLOCKS_PER_SEC`.

Ainsi, `clock()` est principalement utilisée pour chronométrer des portions de code, mais elle est limitée par la résolution des "ticks" d'horloge et le temps CPU (pas le temps réel écoulé).

La fonction sizeof()

L'opérateur `sizeof` en C est utilisé pour connaître la taille en octets (bytes) d'une variable, d'un type de données ou d'une structure. Cet opérateur est évalué **à la compilation**, ce qui signifie que la taille est déterminée avant l'exécution du programme, sans coût de performance à l'exécution.

Syntaxe

```
sizeof(expression ou type)
```

Caractéristiques

— **Temps de compilation** : La taille est déterminée à la compilation.
— **Renvoie un `size_t`** : Le résultat de `sizeof` est de type `size_t`, qui est un type entier non signé (`unsigned int`).
— **Dépend du système** : La taille des types de données (comme `int`, `double`, etc.) peut varier en fonction du système ou de l'architecture (32 bits ou 64 bits).

Exemple Calcul de l'utilisation de la mémoire d'un tableau

```c
#include <stdio.h>

int main() {
        int tableau[10]; // Tableau de 10 entiers

        // Calcul de la taille totale du tableau en octets
        size_t taille_tableau = sizeof(tableau);
        // Calcul de la taille d'un element du tableau
        size_t taille_element = sizeof(tableau[0]);

        // Affichage des resultats
        printf("Taille totale du tableau : %zu octets\n", taille_tableau
            );
        printf("Taille d'un element du tableau : %zu octets\n",
            taille_element);
        printf("Nombre d'elements dans le tableau : %zu\n",
            taille_tableau / taille_element);

        return 0;
}
```

1. Exécuter le programme montré dans l'exemple.

2. Varier les valeurs de n et afficher les temps respectifs en secondes.
3. Modifier le programe pour afficher le temps moyen d'execution pour chaque valeur de n.
4. Modifier le programme pour générer un fichier `.csv` en sortie contenant pour chaque valeur de n le temps de moyen d'exécution.
5. Exporter le fichier en sortie à excel et dresser les courbes à partir du tableau. Comprer le résultat avec la complexité théorique.

Exercice 2.- Tester la primalité d'un nombre.

Soit $n \in \mathbb{N}$. Proposer trois algorithmes pour vérifier si n est premier ou non.

— Varier les valeurs de n et dresser le tableau de complexité pratique pour chacun des algorithmes.

— Expoter les résultats en Excel et comprer les courbes des trois algorithmes.

Exercice 3.- Calcul des sommes.

Soint les expressions des sommes suivantes :

$$\sum_{i=1}^{n} i \qquad (1) \qquad\qquad \sum_{i=1}^{n} i^2 \qquad (2) \qquad\qquad \sum_{i=1}^{n} i^3 \qquad (3)$$

— Proposer deux algorithmes (avec des complexités différentes) pour le calcul de chacune de ces sommes.

— Varier les valeurs de n et dresser le tableau de complexité pratique pour chacun des algorithmes.

— Expoter les résultats en Excel et comprer les courbes des différentes algorithmes.

Exercice 4.- La génération aléatoire des données avec `rand()`.

La fonction `rand()` en C fait partie de la bibliothèque standard `<stdlib.h>` et est utilisée pour générer des nombres pseudo-aléatoires. Chaque appel à `rand()` renvoie un entier qui correspond à un nombre pseudo-aléatoire compris entre `0` et `RAND_MAX`.

Prototype

```
#include <stdlib.h>
int rand(void);
```

Détails

— La fonction renvoie un entier pseudo-aléatoire entre `0` et la constante `RAND_MAX`.
— La valeur de `RAND_MAX` peut varier selon les systèmes, mais elle est souvent égale à 32767.
— `rand()` génère une séquence pseudo-aléatoire qui peut être répétée si la graine (`seed`) n'est pas modifiée.

Initialiser une graine avec `srand()`

La fonction `srand()` permet d'initialiser le générateur de nombres pseudo-aléatoires avec une graine. Si la graine est fixe, la séquence de nombres générés sera toujours la même. Pour obtenir une séquence différente à chaque exécution, on utilise généralement la fonction `time(NULL)` pour initialiser la graine avec le temps actuel.

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Exemple d'utilisation de `rand()` et `srand()`

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

int main() {
        // Initialiser la graine avec le temps actuel
        srand(time(NULL));

        // Generer et afficher un nombre pseudo-aleatoire
        int random_number = rand();
        printf("Nombre aleatoire : %d\n", random_number);

        return 0;
}
```

Générer un nombre dans une plage spécifique

Pour générer un nombre aléatoire dans une plage définie, par exemple entre `min` et `max`, on utilise la formule suivante :

```
int random_number = min + rand() % (max - min + 1);
```

Voici un exemple de génération d'un nombre entre 1 et 100 :

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

int main() {
        srand(time(NULL));

        // Generer un nombre aleatoire entre 1 et 100
        int random_number = 1 + rand() % 100;
        printf("Nombre aleatoire entre 1 et 100 : %d\n", random_number);

        return 0;
}
```

Explication de la fonction `time(NULL)` en C

La fonction `time()` en C est utilisée pour obtenir l'heure actuelle, exprimée en secondes écoulées depuis le **1er janvier 1970** (l'époque Unix, ou "Unix Epoch"). Lorsqu'elle est appelée avec `NULL` comme argument, elle renvoie directement le nombre de secondes écoulées depuis cette date.

Prototype

```
#include <time.h>
time_t time(time_t *t);
```

Détails

— **Argument `NULL` :**
  — Si vous passez `NULL` à la fonction `time()`, elle ignore le paramètre et retourne simplement le nombre de secondes écoulées depuis le début de l'époque Unix.
  — Si un pointeur valide est passé à la fonction, la valeur du temps actuel sera aussi stockée dans l'emplacement pointé par ce pointeur.

— **Valeur de retour :**
  — La fonction retourne un type `time_t`, qui est souvent un entier long représentant le nombre de secondes écoulées depuis l'époque Unix (1er janvier 1970 à 00 :00 :00 UTC).

Exemple d'utilisation de `time(NULL)`

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

int main() {
        // Obtenir l'heure actuelle en secondes depuis l'epoque Unix
        time_t current_time = time(NULL);

        // Afficher le temps en secondes
        printf("Temps actuel (secondes depuis 1970) : %ld\n", current_time);

        return 0;
}
```

Dans cet exemple :
— `time(NULL)` obtient le nombre de secondes écoulées depuis l'époque Unix.
— Le temps est stocké dans la variable `current_time` de type `time_t`, qui est ensuite affichée.

Utilisation typique : graine pour le générateur de nombres aléatoires

L'exemple classique d'utilisation de `time(NULL)` est dans l'initialisation de la graine pour le générateur de nombres pseudo-aléatoires avec la fonction `srand()`.

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

int main() {
        // Initialiser la graine du generateur aleatoire avec le temps actuel
        srand(time(NULL));

        // Generer un nombre aleatoire
        int random_number = rand();
        printf("Nombre aleatoire : %d\n", random_number);

        return 0;
}
```

Remarques

— `time(NULL)` renvoie le nombre de secondes écoulées depuis l'époque Unix, mais n'inclut pas les heures, minutes ou secondes. Pour obtenir des informations détaillées, il est nécessaire d'utiliser `localtime()` ou `gmtime()` pour convertir la valeur en une structure de type `tm`.
— La résolution de `time()` est d'une seconde, ce qui peut ne pas être suffisant pour des applications nécessitant des mesures plus précises.

1. En utilisant la fonction `rand` , écrire une fonction `genererTab` qui permet de générer aléatoirement un tableau de n entiers compris entre 0 et k, $0 \leq T[i] \leq k$.

2. Ecrire la fonction *rechercheSeq* qui effectue une recherche séquentielle de la valeur x dans le tableau T, elle retourne 1 si x se trouve dans T et 0 sinon.

3. Ecrire la fonction $Trier$ qui permet de trier le tableau T.

4. Soit T un tableau trié. Ecrire la fonction *rechercheDicho* qui effectue une recherche dichotomique de la valeur x dans le tableau T, elle retourne 1 si x se trouve dans T et 0 sinon.

5. Evaluer et comprer les complexités des deux fonction de recherches en comparant leurs courbes pratiques et leurs complexités théoriques.

6. Ecrire une fonction `genererMat` qui permet de générer aléatoirement une matrice de(nxm) entre $0etk, 0 \leq M[i][j] \leq k$.

7. Ecrire deux fonction de multiplication de matrices (la méthode usuelle et celle de Strassen). Comparer les complexités de temps et de mémoires des deux méthodes.

Remarque

Il serait intéressant de tester les algorithmes sur différents langages de programmations par exemple : Le langage C, Python et Java, en utilisant la fonctionnalité de l'optimiseur pour chacun.