Programmation Procédurale en langage C – Examen (durée 1h30)

Deuxième session du 22 juin 2026

Consignes pour la composition de l'examen sur feuille
La compilation et l'exécution de vos fonctions et programmes est permise depuis Moodle pour l'exercice 2. Cependant, vos réponses doivent être rendues sur feuille.

Notes : Aucun document autorisé. Sont interdits les calculatrices, les téléphones, ainsi que tout autre ustensile de calcul ou de communication, en dehors de votre session Moodle depuis votre ordinateur ESTIA.

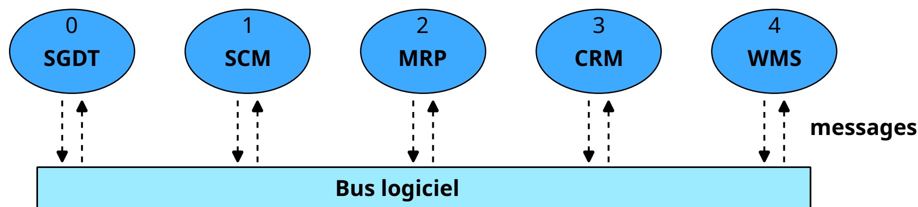
Exercice 1 : Questions de connaissance générale (5 points)

Les réponses aux 5 questions ont déjà été données sur la page Moodle précédente.

Exercice 2 : Une fonction et deux programmes à écrire (15 points)

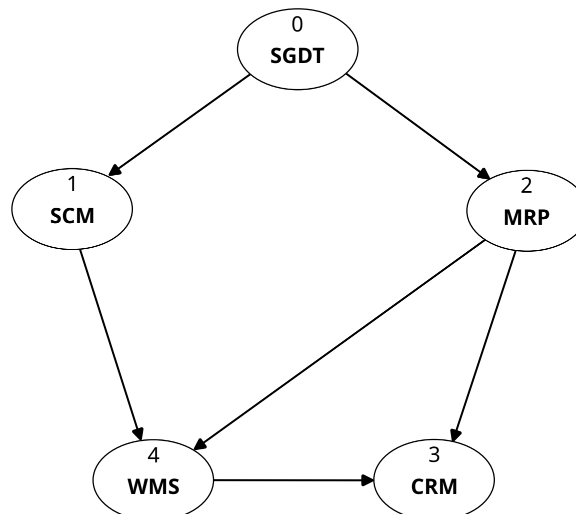
Synchronisation dans une infrastructure logicielle centralisée

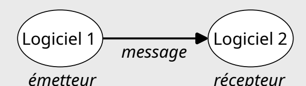
Cet exercice traite de la propagation de messages dans le cas de l'intégration des applications (EAI) d'un système d'information (SI) centralisé autour d'un bus concentrateur (Hub). Ci-dessous un exemple avec cinq logiciels de gestion (numérotés de 0 à 4) couramment retrouvés dans les catalogues des prestataires de solutions de gestion :



Glossaire
SGDT : Système de Gestion de Données Techniques
SCM : Supply Chain Management
MRP : Manufacturing Requirements Planning
CRM : Customer Relationship Management
WMS : Warehouse Management System

La propagation d'une information entre plusieurs logiciels de gestion peut nécessiter de respecter un certain ordre de diffusion pour synchroniser leurs bases de données. Par exemple, voici un exemple de graphe de dépendance entre nos cinq logiciels :





Ainsi, pour respecter les dépendances, lorsque le logiciel SGDT (0) émet une nouvelle information, elle est d'abord diffusée aux logiciels MRP (2) et SCM (1), puis WMS (4), et enfin CRM (3). Un ordre possible de diffusion par le bus logiciel est donc : 0, 2, 1, 4, 3. Remarquons que l'ordre de diffusion n'est pas toujours unique. Par exemple, un autre ordre possible dans notre cas serait : 0, 1, 2, 4, 3.

Pour calculer l'ordre de diffusion des messages dans notre graphe de dépendance (graphe orienté acyclique), nous proposons d'implémenter un algorithme basé sur le tri topologique (lui-même construit à partir de l'algorithme de parcours en profondeur) dans lequel nous intégrons le degré entrant de chaque nœud (c.-à-d., le nombre d'arcs arrivant sur chaque logiciel et le nombre de visites des nœuds lors du parcours. Nous choisissons de nous baser sur la version itérative de l'algorithme de parcours en profondeur, qui nécessite un principe de piles LIFO (Last In First Out). Le graphe de dépendance sera quant à lui implémenté par une matrice d'adjacence.
Le nombre de logiciels est déterminé par la macro-constante suivante :

```
#define N 5
```

1) [2 points] Écrivez une fonction permettant de mettre 0 dans toutes les cases d'un tableau de taille N, en respectant le prototype suivant :

```
void reset(int tab[N]);
```

2) [4 points] Écrivez une fonction permettant de calculer le degré entrant de chaque sommet d'un graphe à partir de sa matrice d'adjacence (de taille $N \times N$), sachant que pour une ligne i et une colonne j, la valeur 1 indique un arc allant du sommet j vers le sommet i, et aucun arc si la valeur est 0, pour des indices i et j dans l'intervalle $[0; 4]$.

Par exemple, la matrice d'adjacence du graphe orienté acyclique présenté ci-avant est :

$$\text{dependance} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Ainsi, les deux valeurs 1 de la première ligne correspondent aux deux arcs sortants du sommet 0 et allant vers les sommets 1 et 2, c'est-à-dire :

— dependance[0][1] vaut 1

— dependance[0][2] vaut 1

De plus, les deux valeurs 1 de la dernière colonne correspondent aux deux arcs arrivant sur le sommet 4 depuis les sommets 1 et 2, c'est-à-dire :

— dependance[1][4] vaut 1

— dependance[2][4] vaut 1

Remarquons aussi que la première colonne est vide car aucun arc n'arrive sur le sommet 0.

Respectez le prototype suivant pour écrire le code de la fonction :

```
void calculer_degre(int dependance[N][N], int entrants[N]);
```

3) [7 points] Nous allons maintenant implémenter notre version itérative du tri topologique, basée sur le nombre de visites des sommets. Le principe de l'algorithme est le suivant :

ALGORITHME

`tri_topologique (dependance: matrice d'adjacence)`

Variables

- `N` : nombre de sommets.
- `visites` : tableau d'entiers de taille `N` qui comptabilisera le nombre de visites de chaque sommet.
- `entrants` : tableau d'entiers de taille `N` qui enregistre le degré entrant de chaque sommet.
- `traitement` : tableau d'entiers de taille `N` utilisé comme une pile de longueur `traitement_longueur` où sont ajoutés les prochains sommets à visiter.
- `tri` : tableau d'entiers de taille `N` utilisé comme une pile de longueur `tri_longueur` où chaque sommet apparaît avant ses successeurs.

```
1  Exécution
2
3  reset(visites);
4
5  calculer_degre(dependance, entrants);
6
7  Pour chaque sommet s Faire :
8    Si s n'admet pas de sommets entrants Alors :
9      ajouter s à la pile de traitement
10   Fin Si;
11 Fin Pour;
12
13 Tant Que la pile de traitement contient des sommets Faire :
14   s = retirer un sommet de la pile de traitement
15   Si le nombre de visites de s est inférieur ou égal au degré entrant de s Alors :
16     incrémenter le nombre de visites de s
17     Si le nombre de visites de s est supérieur ou égal au degré entant de s Alors :
18       ajouter s à la pile de tri
19       Pour chaque sommet successeur i du sommet s Faire :
20         Si le nombre de visites de i est inférieur ou égal au degré entrant de i Alors :
21           ajouter i à la pile de traitement
22         Fin Si;
23       Fin Pour;
24     Fin Si;
25   Fin Si;
26 Fin Tant Que;
27
28 afficher les éléments du tableau tri, du premier au dernier, càd ordre LIFO
29
30 Fin Exécution;
```

Pour gérer les piles de manière simple, les éléments seront enregistrés dans des tableaux d'entiers et une variable entière permettra de compter le nombre d'éléments présents. Les opérations d'empilement et de dépilement d'entiers deviennent alors :

```
int pile[N];
int longueur = 0;
int element;
pile[longueur++] = element ; /* ajouter un element */
element = pile[--longueur] ; /* retirer un element */
```

Une autre possibilité est de définir les macro-fonctions suivantes :

```
#define LIST_PUSH(L, N, E) L[N++] = E
#define LIST_POP(L, N) L[--N]
#define LIST_PRINT(L, N) printf("[");{int i;for (i=0;i<N;i++){\
  printf("%s %d", i>0?",":"", L[i]);}}printf("]\n")
```

Maintenant, écrivez la fonction implémentant l'algorithme, en respectant le prototype suivant :

```
void tri_topologique(int dependance[N][N]);
```

4) [2 points] Appliquez le tri topologique en écrivant un programme qui calcule le tri topologique du graphe de dépendance ci-avant en déclarant sa matrice d'adjacence comme suit :

```
int dependance[N][N] = {
  { 0, 1, 1, 0, 0 },
  { 0, 0, 0, 0, 1 },
  { 0, 0, 0, 1, 1 },
  { 0, 0, 0, 0, 0 },
  { 0, 0, 0, 1, 0 },
} ;
```