Algorithmique et Programmation C (CS101) – Examen Session 1 2022-23

Durée totale : 1 heure 30 / Tiers Temps = 2 heures

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiant·e·s est interdite. Aucun document autorisé (on fournit une fiche de syntaxe C)

— Il est **indispensable de rédiger vos réponses, et d'expliquer vos programmes.** Un programme non expliqué n'aura pas la totalité des points.

— Les questions (notamment du premier exercice) ne **SONT PAS** de difficulté croissante.

— Cet examen est probablement **trop long**, le barème en tiendra compte. Les points donnés dans les questions (pour un total de 22 points) sont des indications de pondération respective (difficulté, longueur) de chaque question.

— Ne pas rendre l'énoncé.

Solution: En rouge, mes commentaires après correction. Vous êtes plus de 60, donc écrire votre nom en MAJUSCULES est indispensable pour que vos enseignant·e·s puissent trier les copies. Je vous laisse imaginer un tri manuel de 60 copies avec des noms illisibles.

---

1 Calcul de puissance

Solution: Les solutions sont en fait des indications de correction qu'il faudrait mieux rédiger.

On désire dans cet exercice calculer x^n, x étant un entier dans $\mathbb{Z}$ quelconque, et n un entier naturel **strictement positif**.

Question #1 (2 points)

Rédiger en C une fonction

```
int puissance_simple(int x, int n);
```

qui calcule x^n de la manière itérative la plus simple possible. Quel est le nombre de multiplications effectuées en fonction de n ?

Solution:

```
int puissance_simple(int x, int n){
  int res=x;
  for (int i = 1; i< n; i++){
    res = res*x;
  }
  return res;
}
```

Cet algorithme (il n'y a pas de y à algorithme) effectue $x \times x \times x$, donc $n-1$ multiplications (d'ailleurs, c'est écrit plus bas).

Attention à ne pas paraphraser le code, si je vois une boucle for, je sais que c'est une boucle for. Privilégier une boucle for à chaque fois que c'est possible. Attention $x = x * x$ ne donne pas le calcul demandé.

Question #2 (2 points)

Donner $P(i)$ une propriété vraie à la fin de la boucle numéro i qui suffit à prouver la correction de l'algorithme précédent. *On ne demande pas la preuve de cette propriété.*

> **Solution:** L'invariant est immédiat "à la fin de la boucle numéro i, res contient x^i. Attention, votre invariant doit être cohérent avec votre programme, si vous commencez votre boucle à 0, il y a peut être un décalage à calculer.

La page Wikipedia fr donne un algorithme dit "d'exponentiation rapide". La figure ci-dessous est une copie d'écran de la page https://fr.wikipedia.org/wiki/Exponentiation_rapide au 23/12/2022.

Soit n un entier strictement supérieur à 1, supposons que l'on sache calculer, pour chaque réel x, toutes les puissances x^k de x, pour tout k, tel que $1 \leq k < n$.

- Si n est pair alors $x^n = (x^2)^{n/2}$. Il suffit alors de calculer $y^{n/2}$ pour $y = x^2$.
- Si n est impair et $n > 1$, alors $x^n = x(x^2)^{(n-1)/2}$. Il suffit de calculer $y^{(n-1)/2}$ pour $y = x^2$ et de multiplier le résultat par x.

Cette remarque nous amène à l'algorithme récursif suivant qui calcule x^n pour un entier strictement positif n :

$$\text{puissance}(x,\, n) = \begin{cases} x, & \text{si } n = 1 \\ \text{puissance}(x^2,\, n/2), & \text{si } n \text{ est pair} \\ x \times \text{puissance}(x^2,\, (n-1)/2), & \text{si } n \text{ est impair} \end{cases}$$

En comparant à la méthode ordinaire qui consiste à multiplier x par lui-même $n-1$ fois, cet algorithme nécessite de l'ordre de $O(\log n)$ multiplications et ainsi accélère le calcul de x^n de façon spectaculaire pour les grands entiers.

Question #3 (2 points)

Exécuter cet algorithme pour les paires (x, n) suivantes : $(3, 4)$ $(3, 17)$. *Bien montrer les appels récursifs. Il n'est pas nécessaire de calculer explicitement les puissances de 3 "temporaires".*

> **Solution:**
>
> 1. pour $(x, n) = (3, 4)$: n est pair, donc l'appel `puissance(x,n)` va lui même réaliser l'appel `puissance(3*3, 2)`, qui va appeler `puissance(3*3*3*3, 1)`, qui retourne $3 * 3 * 3 * 3 = 3^4$. Les retours des appels récursifs sont immédiatement copiés vers l'appelant, et finalement le résultat est 3^4.
> 2. pour $(x, n) = (3, 17)$, on va schématiser autrement :
>
> ```
> puissance(3,17)--> 3 * puissance(3*3, 8)
>  (appel rec, PUIS calcul de la multiplication, plus tard)
>                --> 3 * (puissance(3*3*3*3, 4))
> (appel rec sans multiplication supplémentaire)
>                --> 3 * (puissance(3*3 .... *3 , 2))
> (dans les ... il y a un calcul de 3^4 au carré, ie 3^8)
>                --> 3 * puissance(3*3 ... * 3, 1)
> (dans les ... il y a un calcul de 3^8 au carré, ie 3^16
>                ---> 3 * (3*3 ...* 3)     (le dernier appel a retourné 3^16)
> ```
>
> Après remontée des appels récursifs on obtient bien 3^{17}. Attention à ne pas dire que l'on retourne (ou renvoie) tel résultat trop tôt... Autre chose, si vous obtenez une puissance qui n'est pas 17, il y a probablement un problème !

Question #4 (1 point)

Traduire cet algorithme en une fonction C récursive. *On rappelle la syntaxe du modulo en C : `u%d` calcule le reste de la division euclidienne de u par d.*

```c
int puissance_rec(int x, int n);
```

> **Solution:** La traduction est immédiate :

```
int puissance_rec(int x, int n){
  if (n==1) return x;
  else {
    if (n%2 == 0) return puissance_rec(x*x, n/2);
    else return x*puissance_rec(x*x, (n-1)/2);
  }
}
```

Attention à faire une disjonction de cas. J'ai peu enlevé de points, néanmoins, à celleux qui ont traduit comme la définition.

Un·e utilisateur·rice ne lisant pas la documentation lance le programme C de la façon suivante : `puissance_rec(3,-2)`. L'exécution dure un peu longtemps, et à la fin termine avec une erreur mémoire. Pour comprendre cette erreur, on regarde la documentation du modulo "possiblement négatif" en C : celui-ci est en fait défini pour toute paire d'entiers naturels a, b (b non nul, a, b peuvent être négatifs). La documentation dit juste que l'égalité suivante est toujours vraie

```
a == ((a / b) * b) + (a % b)
```

avec / la division entière avec troncature vers 0 .

Question #5 (2 points)

Justifier la valeur `-1%2=-1` en C, et conclure.

Solution: On prend a=-1 et b=2. alors la division -1/2 avec troncature vers 0 vaut 0, donc `a%b` vaut -1.
Attention au raisonnement par équivalence avec égalité ; cela devient rapidement faux.
Appel rec "à l'infini", et au bout d'un moment il y a dépassement de capacité car la pile des appels récursifs est trop grosse.
J'ai mis tous les points à "appels récursifs à l'infini.

Pour la suite, on reprend notre puissance définie avec $n > 0$.

Question #6 (1 point)

On fournit ci-dessous une version itérative de l'algorithme :

```
int puissance_rapide(int x, int n) {
    int p = x;
    int r = 1;

    while (n > 0) {
        if (n % 2 == 1) r = r*p;
        p = p * p;
        n = n/ 2;
    }
    return(r);
}
```

Exécuter `puissance_rapide(3,17)`.

Solution: Initialement les valeurs de (p, r, n) sont $(3, 1, 17)$, et ensuite on exécute la boucle "tant que" : (et on donne les valeurs à la fin de chaque boucle) $(3*3, 3*1; 8)$, $(3*3*3*3, 3, 4)$, $(3^8, 3, 2)$, $(3^{16}, 3, 1)$, $(3^{32}, 3*3^{16}, 0)$, et la valeur retournée est bien 3^{17}.
oui, la division est bien la division entière, comme depuis le début de l'exercice. Attention à ne pas aller trop vite, la dernière étape avec n=1 est indispensable pour comprendre ce qui se passe, et ne pas renvoyer 0.

On admet maintenant que les deux implémentations (récursive, itérative), effectuent le même calcul et le même nombre d'étapes.

Question #7 (1 point)

Justifier le texte de la page Wikipédia "cet algorithme nécessite de l'ordre de $O(\log n)$ multiplications". *On pourra raisonner pour les puissances de 2 : $n = 2^k$.*

Solution: L'algorithme pour les puissances de 2 n'effectue jamais la multiplication supplémentaire (dans le if), mais uniquement des divisions successives de n par 2. Quand n vaut 2^k, il y a k divisions, c'est à dire $k = \log_2(n)$. Il ne suffit pas de dire que lorsque $n = 2^k$ $k = log(n)$, il faut évidemment se rapporter au programme analysé !

Question #8 (1 point)

Avec un exemple bien choisi, montrer que cet algorithme n'effectue pas toujours le nombre minimum de multiplications possibles.

Solution: Prendre n=15, cet algorithme effectue 6 multiplications, et il existe une façon de calculer avec 5 multiplications :

- On calcule x^2 (1 multiplication), puis x^3 (1 supplémentaire)
- On calcule x^5 à l'aide des deux résultats précédents (1 multiplication supplémentaire)
- On calcule x^{10} en mettant le précédent au carré (1 multiplication supplémentaire)
- On calcule $x^{15} = x^{10} * x^5$. En tout donc, 5 multiplications.

Le·a lecteur·rice intéressé·e pourra chercher "power tree".

Alors, la conceptrice du sujet aurait du préciser $x, n > 2$. Je voulais qu'on se creuse les méninges... Du coup, mon exercice est beaucoup moins drôle, et vous êtes allé·e·s chercher des cas avec $n = 1$ qui fonctionnent. Snif.

Question #9 ()

(Bonus) Donner un invariant de boucle pour la version itérative.

Solution: Pas tout à fait simple, c'est $x^{n0} = r \times p^n$ avec n_0 la valeur initiale de n. Bravo à un certain ZC qui est le seul à avoir trouvé cet invariant !

L'énoncé du deuxième exercice rédigé se trouve sur la page suivante

2 Représentation d'ensembles d'entiers par tableau

Les sections ci-dessous sont indépendantes. Dans toutes les questions, vous avez le droit de faire appel aux fonctions des questions précédentes, même si vous ne les avez pas écrites.

2.1 Représentation par tableau de booléens

On considère un ensemble d'entiers dont la valeur est entre 0 (compris) et `MAX-1` (compris), `MAX` étant une constante fixée à l'avance. Cet ensemble est représenté par un tableau de booléens e : $e[i]$ à `true` indique que l'entier i appartient à l'ensemble, $e[i]$ à `false` indique qu'il n'appartient pas à l'ensemble, par exemple, le tableau :

i	0	1	2	3	5	6	7	...	21	22	..	MAX-1
$e[i]$	true	false	true	false	false	true	false	...	true	false	...	false

représente l'ensemble $\{0, 2, 6, 21\}$. On appelle cette représentation **représentation de type 1**.

Question #10 (1 point)

Ensemble vide : écrire une procédure

```
void ensembleVide(bool t[MAX])
```

qui prend en paramètre le tableau t et qui le modifie pour qu'il représente l'ensemble vide ($\{\}$).

Solution:

La procédure prend en paramètre un tableau t uniquement. Dans l'encodage proposé, le tableau t encode l'ensemble vide si et seulement si toutes ses cases contiennent le booléen `false`. On effectue donc un parcours de t en réalisant l'affectation $t[i] = false$ pour tout i :

```
void ensembleVide(bool t[MAX]) // aucun element = tout à faux.
{
  int i;
  for (i=0;i<=MAX-1;++i)
    {
      t[i] = false;
    }
}
```

Cette question est une question d'initialisation, donc on ne peut pas faire l'hypothèse que le tableau contient des valeurs true ou false à l'appel de cette fonction, le tableau est à écriture seulement.

Remarquer que les constantes booléennes ne sont pas des chaînes, donc pas de quote, et pas de guillemets.

Question #11 (1 point)

Écrire une fonction

```
int nombreElements(bool t[MAX])
```

qui retourne le nombre d'éléments (ou cardinal) de l'ensemble représenté par le tableau t.

Solution:

Dans l'encodage 1, le nombre d'éléments d'un ensemble est le nombre de cases i dont la valeur $t[i]$ est à *true*. On parcourt donc le tableau t en incrémentant un compteur entier à chaque fois que c'est le cas. On utilise une fonction pour retourner cet élément :

```
int nombreElements(bool t[MAX])
{
  int i,cpt;
  cpt = 0;
  for (i=0;i<=MAX-1;++i)
    {
      if (t[i]) ++cpt; // increment si true.
    }
  return cpt;
}
```

Remarquez que `t[i]==true` est la même chose (ie, a la même valeur booléenne) que `t[i]`. "Il fait beau est vrai" est la même chose que "il fait beau".

Question #12 (1 point)

Écrire une procédure d'affichage d'un ensemble, sur l'exemple la procédure d'affichage écrira `{0,2,6,21}` (avec les virgules et les accolades) sur le terminal. *Si vous avez une virgule en trop, ce n'est pas le drame.*

Solution:
Sur l'encodage 1, il s'agit d'afficher les indices i tels que $t[i]$ vaut *true*. Les accolades sont affichées en dehors de la boucle de parcours. Pour mettre le bon nombre de virgules, on imprime les éléments précédés d'une virgule, sauf le premier élément (le booléen `vide` est à faux dès que l'on a rencontré un élément de l'ensemble) :

```
void afficheEnsemble(bool t[MAX])
{
  int i;
  bool vide=true;
  printf("{");
  for (i=0;i<=MAX-1;++i)
    {
      if (vide && t[i])
            {printf("%d",i);vide=false;} // sans virgule
      else
            if (t[i]) {printf(",%d",i);};
    }
  printf("}\n");
}
```

Question #13 (2 points)

Programme principal Écrire (toujours en C) un programme complet (`main.c`) qui :

- Déclare une constante symbolique `MAX` fixée à 22.
- Déclare et initialise un tableau `ens1` pour qu'il représente l'ensemble $\{0, 2, 6, 21\}$.
- Affiche cet ensemble.
- Calcule et imprime le cardinal de cet ensemble.

Solution: Sans trop de difficulté.

```
#define MAX 22
int main(){
  bool ens1[MAX];
  ensembleVide(ens1);
  ens1[0]= true;ens1[2]=true;
  ens1[6]= true;ens1[21]=true;
```

```
  afficheEnsemble(ens1);
  printf("card=%d\n",nombreElements(ens1));

  return 0;
}
```

Dans cette question, trop de mauvais passages de paramètre tableaux (alors que c'est dans la feuille de syntaxe C !). De même, attention à bien récupérer le RESULTAT renvoyé par nombreElements. Enfin il est bien évident qu'on parle toujours d'un tableau représentant un ensemble de type 1, donc un tableau de booléens.

2.2 Changement de représentation

L'ensemble $\{0, 2, 6, 21\}$ peut aussi être réprésenté par le tableau suivant `tab` : (**type 2**)

i	0	1	2	3	5	6	7	8	...	MAX-1
$tab[i]$	0	2	6	21	-1	-1	-1	-1	...	-1

Les premières cases du tableau contiennent les éléments de l'ensemble **pas forcément dans l'ordre croissant** et les autres contiennent -1.

Question #14 (1 point)

Quel est le coût de la recherche de l'appartenance d'un élément à un ensemble représenté "type2" ?

Solution: Au pire le cardinal de l'ensemble, car il faut faire un parcours jusqu'à rencontrer le premier -1. O(n) ne veut rien dire si n n'existe pas, encore moins O(i).

Question #15 (2 points)

Décrire et implémenter en C un algorithme de conversion d'un ensemble représenté par un tableau **du type** 2 en un tableau qui représente le même ensemble mais **de type** 1. *Des réponses partielles à base d'exemples seront lues avec bienveillance*

Solution: La procédure que nous allons décrire prendra en paramètre deux tableaux : un tableau `type2` d'entiers de taille MAX représentant l'ensemble d'entrée sous l'encodage de type2, et un tableau de booléens de même taille qui servira à stocker le résultat de la conversion et qui est aussi passé en paramètre.

Dans cette question comme dans la suivante, trop de personnes imaginent que l'on peut retourner un tableau. Ce n'est pas possible en C. On passe donc un tableau "à remplir" comme deuxième argument de la fonction.

Maintenant, pour passer de la représentation de type 2 à la représentation de type 1, par exemple sur le tableau représentant l'ensemble $\{0, 2, 6, 21\}$ avec l'encodage 2, on parcourt ledit tableau jusqu'à ce que la case `type2[i]` soit égale à -1 (ou la fin de ce tableau), et pour chaque élément `type2[i]` différent de -1, on met la case correspondante à `true` dans le tableau `type1`. Enfin, on fait attention au fait que le tableau `type1` doit contenir uniquement des `true` et des `false`. Attention aux différences de codage des deux tableaux considérés. Il peut être judicieux de nommer avec des suffixes 1 ou 2. t et T sont des mauvaises idées.

On obtient donc :

```
void conversionVersType1(int type2[MAX], bool type1[MAX])
{
  int i=0;
  ensembleVide(type1);
  while(i<=MAX-1 && type2[i]!=-1)
    {
      type1[type2[i]]=true;
```

```
      ++i;
    }
}
```

Dans cet algorithme, une seule boucle suffit pour avoir l'information voulue. J'ai accepté certaines solutions compliquées à base de deux boucles imbriquées, mais c'est parce que nous sommes en 1A. Les algorithmes/programmes non expliqués n"ont pas eu tous les points.

Question #16 (2 points)

(+ difficile) Idem, dans l'autre sens.

Solution:

Cette conversion est un peu plus compliquée. Elle se déroule en deux phases :

— Dans un premier temps, on parcourt le tableau de type 1 en ajoutant au deuxième tableau la valeur i lorsque `type1[i]` est à vrai. Un curseur (nommé cpt) donne la première case disponible dans le tableau 2.

— Dans un deuxième temps, toutes les cases restantes du tableau de type 2 sont remplies par des -1.

```
void conversionVersType2(bool type1[MAX], int type2[MAX])
{
  int i; // parcours du tableau 1
  int cpt=0; // curseur sur le tableau 2

  for (i=0;i<=MAX-1;++i) // je parcours tout le tableau type1
    {
      if (type1[i]) // si i est element de l'ensemble
        {
         type2[cpt]=i; // ajout de i dans la case courante de type2
         ++cpt;
        }
    }
  // on met des -1 partout ensuite dans type2.
  while (cpt<=MAX-1)
    {
      type2[cpt]=-1;
      ++cpt;
    }
}
```

Le remplissage du tableau de type 2 avec des -1 partout peut se réaliser avec un parcours entier du tableau au début, aussi. Remarquez que j'appelle les tableaux type1 et type2, c'est leur nom, pas leur type.

Solution: Table des points

Question:	1	2	3	4	5	6	7	8	9
Points:	2	2	2	1	2	1	1	1	0
Score:									

Question:	10	11	12	13	14	15	16		Total
Points:	1	1	1	2	1	2	2		22
Score:									

1 Premier programme

```c
/* Source code (hello.c) */
#include <stdio.h>

int main(){
  printf("Hello World!\n");
  return 0;
}
```

Compilation : dans un *terminal* :

```
clang-14 hello.c -o hello
```

On peut remplacer `clang` par `gcc` et ajouter les options `-Wall` ou `-Werror`.

Exécution : si la compilation réussit (sans erreur), on peut alors exécuter ce binaire, toujours dans un terminal, avec `./hello`.

2 Structure générale d'un programme

(ici, un seul fichier source, les seules inclusions sont des entêtes de bibliothèques standard)

```c
#include <stdio.h> // en-têtes de bibliothèques (sans ;)
#define TAILLE 100 // constante symbolique

int c;                          // définitions de variables globales (optionnel)

int mafonction(char u){         // définitions de fonctions (optionnel)
  ...
  return -42;
}

int main(){                     // fonction principale du programme (obligatoire)
  mafonction('u');
  ...
  return 0;                                  // ou EXIT_SUCCESS
}
```

3 Types de données

- **Types de base :** `int` (entier signé), `char` (caractère, sur 1 octet), `bool` (avec `stdbool.h`)
- **Type composé :** les tableaux statiques : `int tab[12]` par exemple.

4 Boucles for

Ces boucles sont à utiliser en priorité lorsque l'on connaît à l'avance le nombre d'itérations.

Exemple : somme des éléments de 1 à 12 :

```c
int i;
int s = 0;
for (i=1; i<=12; i++) // attention aux ;
  s = s +i ;
```

S'il y a plusieurs instructions dans le *corps*, alors les accolades sont utiles :

```c
for (int j=12; j>=0; j--) {// boucle décroissante
  // instruction1
  // instruction2
}
```

5 Boucles while

Lorsque la condition est plus complexe, et qu'on ne peut pas facilement borner le nombre d'itérations :

```c
int c = 0;
while(b > 1){
  b = b/2;
  c++;
} // calcule le log_2 de b.
```

La condition du while, qui est une *condition booléenne* (s'évalue à vrai/faux) peut être composée avec non (`!`), ou (`||`), et (`&&`). Attention aux parenthèses.

6 Fonctions

Une fonction sert à créer une portion de code que l'on peut appeler plusieurs fois.

Déclaration et implémentation d'une fonction *nommée* `mafonction`, qui a un unique *paramètre* de type caractère, et qui renvoie ("retourne") un entier (son *type de retour* est `int`) :

```c
int mafonction(char c){
  return -42;
}
```

Une fonction peut ne rien retourner, le type de retour est alors `void`. Une fonction sans paramètre (appelée *procédure*) a `void` tout seul entre les parenthèses. (En cas de parenthèses "sans `void`", elle accepte n'importe quel nombre de paramètres). Les modifications de paramètres (sauf tableaux) ne sont pas enregistrées à l'extérieur de la fonction.

Appel d'une fonction (à l'intérieur d'une autre fonction, le main, par exemple) :

```c
int resu; // variable du type de retour
resu = mafonction('u'); //le paramètre formel est remplacé par une valeur
```

Alternativement, on peut utiliser le résultat directement :

```c
printf("Le resultat est = %d", mafonction('Z')); //%d car le résultat est un entier (int)!
```

7 Tableaux

Déclaration d'un tableau :

```c
int t[12]; // comme variable locale d'une fonction
```

ou alors, avec une constante symbolique

```c
#define N 100 // au début du programme
...
int tab[N]; // quelque part dans une fonction
```

⚠ Le tableau n'est **pas** automagiquement initialisé. par contre :

```c
int tab[23]={1,12}; // le reste des cases est initialisé à 0.
```

Lecture, écriture

```c
t[2] = ...; // écriture dans la 3ième cellule
... = t[9]; // lecture de la 10ième cellule
```

Passage de paramètre tableau

```c
imprime_tableau(t); // et pas t[N] !
```

⚠ Toute modification du contenu tableau est enregistrée.

Tableaux de caractères Pas de "string" en C, mais des tableaux de `char` avec une sentinelle :

```c
// déclaration et initialisation
char s[21]="hello"; // ajoute `\0` à la fin
```

On peut utiliser les fonctions de la bibliothèque `string`, ou parcourir comme ceci :

```c
while(s[i] != '\0') {...
```

8 Entrées-sorties

Sortie : impression sur le terminal (printf a un nombre de paramètres variable) :

```c
printf("%d", x);// imprime la valeur entière contenue dans x
printf("Ou alors %c, %s\n", caractere, chaine);// autant de % que de variables à imprimer
```

Le premier argument est appelé *chaîne de formattage*.

Entrée : récupération d'une information entrée au clavier :

```c
scanf("%d",&x); // modifie la valeur de x (int) avec la valeur entrée au clavier
scanf("%c",&c);
scanf("%s", s); // une chaîne est un tableau, pas de &
```