Algorithmique et Programmation C (CS101) – Examen Session 1 – 2025-26

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

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiant·e·s est interdite. Calculatrice interdite. Aucun document autorisé (sauf votre 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 ne **SONT PAS** de difficulté croissante.
— Le barème (sur 20 points) est indicatif.
— Ne pas rendre l'énoncé.

Solution: Les solutions proposées sont des éléments de correction qu'il faudrait rédiger d'avantage. Le barème finalement appliqué est celui-ci :

Note Exo 1	Exo 2						Note Exo2	Exo3				Note Exo3
	comprehension (Q1,2)	enTete	apparaitV1	compteocc	Q6main	ncomp		freqlettres	sousmots2	affichefreq2	compl	
5	2	2	2	2	3	1	12	2	1	2	1	6

et la note sur 23 a finalement constitué la note sur 20.
En rouge, mes commentaires après correction.

Les algorithmes demandés seront écrits en C.

Exercice 1 – Programme mystère (20 min)

Voici un programme C (à lire colonne par colonne) :

```c
#include <stdio.h>
#define N 3

void display (int t[N]){
  int i;
  for(i=0;i<N;i=i+1){
    printf("%d␣",t[i]);
  }
  printf("\n");
}

void swap3 (int t[N],int i,int j){
  int tmp = t[i];
  t[i]=t[j];
  t[j]=tmp;
}

void init(int tt[N]){
  int i;
  for (i=0;i<N;i=i+1){
    tt[i] = i+1;
  }
}
```

```c
void mysteriousRec(int tab[N],int k) {
  int j;
  if ( k == N-1 ) display (tab);
  for (j = k ; j < N ; j ++) {
    swap3 (tab,k,j);
    mysteriousRec(tab,k+1);
    swap3 (tab,k,j);
  }
}

int main(){
  int montab[N];
  init(montab);

  mysteriousRec(montab,0);
  return 0;
}
```

Question #1.1 (2 points)
Quelles impressions sont effectuées par ce programme compilé puis exécuté sur le terminal ? *quelques lignes*

Question #1.2 (1 point)
Pourquoi l'appel `mysteriousRec(t,0)` termine-t-il ?*quelques lignes*

Question #1.3 (2 points)
Que fait ce programme dans le cas général ? (N quelconque). Justifier *10 à 15 lignes*.

Solution: Ci dessous des éléments de correction qu'il faudrait développer. L'initialisation du tableau : $[1, 2, 3]$ Cela imprime :

```
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

(donner des étapes) L'appel termine car chaque appel récursif se fait avec un deuxième paramètre entier croissant (k), qui finit par atteindre la valeur $N-1$, pour laquelle la fonction retourne immédiatement.

Dans le cas général, cette fonction réalise toutes les permutations de l'ensemble $\{1, \dots N-1\}$, qu'elle imprime sur le terminal, en se basant sur l'observation suivante : pour réaliser une permutation de $\{1, \dots k\}$, on peut :

— énumérer toutes les possibilités pour le premier élément. (d'où la boucle for sur j)

— réaliser les permutations sur le reste

L'échange après l'appel récursif est pour "remettre le tableau dans l'état précédent".

Cet exercice a été corrigé par JLK.

Avec le barème initial, cet exercice a eu un score de $0,8/5$ avec un écart type de $0,98$.

Les deux exercices suivants sont basés sur la même entrée, à savoir un texte, considéré stocké dans un tableau `texte`, de taille N **constante globale** (pour les exemples, $N = 13$). Le tableau est considéré rempli (toutes les cases de 0 à $N-1$ contiennent donc un caractère) par des caractères de 'a' à 'z'. L'objectif est de pouvoir rechercher un mot quelconque dans ce texte, c'est-à-dire pouvoir dire si ce mot apparaît dans le texte, à quel endroit, combien de fois.

Attention, aucune des fonctions de string.h ne pourra être utilisée !

i	0	1	2	3	4	5	6	7	8	9	10	11	12=N-1
$texte[i]$	q	u	e	l	b	o	n	b	o	n	b	o	n

On considère que les mots sont plus petits que le texte, les mots sont donc stockés comme d'habitude dans un tableau de taille N aussi, mais avec le caractère de fin $\backslash 0$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12 =N-1
$mot1[i]$	b	l	a	b	l	a	$\backslash 0$	—	—	—	—	—	—

Exercice 2 – Recherche d'un mot complet (50 min)

Definition 1. *Un mot* `mot` *(de taille m) apparaît dans un texte* `texte` *si il existe un indice i tel que le sous-tableau* `texte[i..i+m-1]` *est égal à* `mot` *(lettre par lettre !).*

Definition 2. *Le* suffixe *numéro k (k entre* 0 *et* $N-1$*) du tableau* `texte` *est le sous-tableau* `texte[k..N-1]`.

Un mot de taille m apparaît donc dans un texte à l'indice k si il apparaît au début du suffixe numéro k. On dit alors que k est un *indice d'apparition*.

Solution: Il n'est pas possible de faire des grossières erreurs de syntaxe lorsque la feuille de syntaxe C fait partie de l'énoncé.
Cet exercice a été corrigé par LG. Avec le barème initial, cet exercice a eu un score de $5,95/10$ avec un écart type de 3.

Question #2.1 ($1/2$ point)
Quel est le suffixe numéro 3 du tableau `texte` contenant la chaîne "quelbonbonbon" (celui qui est dessiné) ?

Solution:
`lbonbonbon`

Question #2.2 ($1/2$ point)
Quels sont les indices d'apparition du mot "bon" dans `texte` ?

Solution: "bon" apparaît 3 fois : aux indices 4, 7, 10.

Question #2.3 (1 point)
Écrire une **fonction C** nommée `enTetedeSuffixe` qui étant donnés `texte`, `mot` (et sa taille `m`) et un indice k, retourne `true` si le mot apparaît à l'indice k, `false` sinon.

Solution:
Dans la suite de ce corrigé, les fonctions sont écrites avec des boucles while sans rupture prématurée du flot, mais la correctrice (LG) a accepté des versions avec rupture prématurée du flot La fonction retourne un booléen, et parcourt le mot tant que les lettres du mot sont égales à celles du texte (décalées de k) :

```
bool enTetedeSuffixe(char texte[N], char mot[N], int m, int k)
{
  bool resu = true;
  int i=0;
  while(resu && i<m)
    {
      if (mot[i]!=texte[k+i])
          resu = false;
      i++;
    }
  return resu;
}
```

Attention, en C on ne peut pas comparer des tableaux. Par ailleurs, Une double boucle est ici inutile, car no sait comment calculer les 2 indices à comparer l'un à partir de l'autre.
Cette correction suppose que le mot ne "déborde pas du texte"

Question #2.4 (2 points)
Écrire une **fonction C** nommée `apparaitV1` qui étant donnés `texte`, `mot` (et sa taille `m`), retourne `true` si le mot apparaît dans le texte, `false` sinon.

Solution: Pour savoir si un mot apparaît, on regarde successivement si il apparaît en tête de chaque suffixe. Les indices des suffixes varient de 0 à $N - m$.

```
\begin{lstlisting}[language=cplus]
bool apparaitV1(char texte[N],char mot[N],int m)
{
  int dec = 0;
  bool trouve = false;

  while(!(trouve) && dec < N-m)
    {
      trouve = enTetedeSuffixe(texte,mot,m,dec);
      dec ++;
    }
  return trouve;
}
```

Trop de copies ne remarquent pas qu'on peut utiliser la fonction précédente !

Question #2.5 (2 points)

Écrire une **fonction C** qui compte le nombre d'occurrences d'un mot donné dans un texte, et qui au passage affiche les indices d'occurences.

Solution: Cette fonction est très similiaire à la précédente, mais cette fois on observe tous les décalages :

```
int nbOccurences(char texte[N],char mot[N],int m)
{
  int dec = 0;
  int  nbocc = 0;

   while(dec <= N-m)
    {
      if  (enTetedeSuffixe(texte,mot,m,dec))
        {
           printf("trouvé à l'indice %d\n",dec);
           nbocc++;
        }
      dec ++;
    }
   return nbocc;
}
```

L'énoncé n'étant pas suffisamment clair sur le type de retour, la correctrice a accepté un type de retour void – avec une impression de nbocc à la fin de la fonction

Question #2.6 (3 points)

Écrire un programme **C complet** (main, fonctions -déclarations, et code avec des pointillés- , etc) :

- Inclusion des librairies `stdio.h` et `stdbool.h`.
- Déclaration de la constante N avec la taille adéquate.
- Déclaration du tableau `texte` et remplissage avec la chaîne "quelbonbonbon".
- Déclaration du tableau `mot` et remplissage avec la chaîne "bon".
- Impression de tous les indices d'occurence du sous-mot `bon` dans `texte`, ainsi que le nombre d'occurences.

Solution: Voici le programme C complet :

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define N 13

//ici toutes les fonctions écrites avant.

int main(){
  char texte[N] = "quelbonbonbon";
  char mot[N]="bon";
  printf("nb occurrences = %d\n",nbOccurences(texte,mot1,3));

  return 0;
}
```

suivant les signatures des fonctions précédentes, il était possible de faire `mot[3]`, ou pas.

Question #2.7 (1 point)

Combien de comparaisons cet algorithme de recherche fait-il au pour un mot de taille m recherché dans un texte de taille N ?

Solution: Quel que soit le mot d'entrée (de taille m), l'algorithme fait $N - m$ fois appel à la fonction `enTetedeSuffixe`. Ensuite, le nombre de comparaisons de cette fonction dépend du mot :

- Si la première lettre du mot n'apparaît jamais dans le texte, à chaque fois la fonction `enTetedeSuffixe` fait une unique comparaison.
- Si le mot complet apparaît à chacun des décalages, on fait m comparaisons par décalages.

On a donc au mieux $O(N)$ comparaisons et au pire $O(N * m)$.

Exercice 3 – Recherche de petits (sous-) mots (20 min)

Definition 3. *La chaîne de caractères associée à un sous-tableau (du texte)* `texte[i..i+m-1]` *est appelée sous-mot (de taille m)*

Dans cette partie, nous nous restreignons aux sous-mots de taille 1 et 2. Nous désirons lister les fréquences d'apparition de ces sous-mots dans `texte` (le nombre de fois où ils apparaissent).

Par exemple, voici les sous-mots de taille 1 de "quelbonbonbon", par ordre alphabétique, avec leur fréquence d'apparition : b(3), e(1), l(1), n(3), o(3), q(1), u(1).

Solution: Avec le barème initial, cet exercice a eu un score de 1,31/5 avec un écart type de 1,19.

Question #3.1 (1½ points)

Ecrire une procédure `void frequenceLettres(texte,tabresu)` qui calcule le nombre d'apparitions de chacune des lettres de l'alphabet dans le tableau `texte`. Cette fonction modifie le tableau `tabresu` de taille 26.

Solution:
On a déjà vu plusieurs fois cette fonction, que l'on appelle histogramme habituellement. Le nombre de 'a' est compté dans la case d'indice 0, le nombre de 'b' dans la case d'indice 1,

etc. On utilise donc un décalage de taille 'a'. il n'est absolument pas nécessaire de connaître le code ascii de 'a'

```
void frequenceLettres(char texte[N], int tabresu[26]){
  for (int i=0;i<26;i++) tabresu[i]=0;

  int j=0,dec;
  while (texte[j] != '\0' && j<N){
    dec = texte[j]-'a'; // a est compté dans la case 0!
    tabresu[dec]++;
  }
}
```

Question #3.2 (1/2 point)

Quels sont les sous-mots de taille 2 apparaissant dans `texte` ("quelbonbonbon") ? Quels sont leurs fréquences d'apparition ?

Solution: A l'aide de la fonction qui suit :

```
le texte est quelbonbonbon
le mot qu apparaît 1 fois
le mot ue apparaît 1 fois
le mot el apparaît 1 fois
le mot lb apparaît 1 fois
le mot bo apparaît 3 fois
le mot on apparaît 3 fois
le mot nb apparaît 2 fois
```

Question #3.3 (2 points)

Écrire une action `void afficheFrequences2(texte)` qui affiche tous les sous-mots de taille 2 ainsi que leur fréquence d'apparition. On fera un algorithme simple, même si il coûte cher. L'algorithme sera bien expliqué avec dessins et exemples.

Solution: Une solution simple (mais coûteuse) consiste à bâtir un peu comme dans l'exercice précédent.

```
void afficheFrequences2bis(char texte[N]){
  // version où l'on affiche uniquement 1 fois l'information
  int nb;
  char tmp[2];
  for (int i=1; i<N;i++){// parcours du texte pour énumérer
//tous les sous-mots de taille 2, dans tmp
    tmp[0]=texte[i-1];
    tmp[1]=texte[i];// copie du mot de taille 2.
    nb=nbapparitions2(texte, tmp);// <--- fonction 1

    if (!apparaitGauche(texte, i, tmp)) //<---- fonction 2
      printf("le mot %s apparaît %d fois\n", tmp, nb);
  }
}
```

La premiere fonction compte le nombre d'occurrences du mot tmp, ce qui n'est pas très difficile :

```c
int nbapparitions2(char texte[N], char mot[2]){
  //ceci est une version plus simple d'une fonction de l'exo 2
  // coûte 1 parcours du texte : O(N) comparaisons
  int j=1;
  int cpt = 0;
  while(texte[j] !='\0' && j < N){
    // attention aux indices
    if (texte[j]==mot[1] && texte[j-1]==mot[0])
      cpt++;
    j++;
  }
  return cpt;
}
```

On peut en rester là, mais les mots de taille 2 seraient affichés plusieurs fois. On décide de détecter (fonction 2) si le mot courant a déjà été vu dans l'énumération du texte. Cela se fait en réalisant la fonction suivante :

```c
bool apparaitGauche(char texte[N], int indexmax, char mot[2]){
  for (int j=1; j <= indexmax-1; j++){// à gauche strictement
    if (texte[j]==mot[1] && texte[j-1]==mot[0]){
      return true;
    }
  }
  return false;
}
```

et voilà.

à rédiger mieux Une autre solution consiste à énumérer tous les sous mots de taille 2, l'avantage étant qu'il n'y a ainsi pas de doublon.

```
......
char mot[2]; // pas forcément utile si on garde i,j c'est bon aussi
int nb;
for i='a', i<'z', i++
   mot[0]=i;
   for j='a'; j < 'z', j++
      mot[1]=j
      // compter le nb d'occ du mot dans le texte supposé rempli
      nb = 0;
      for (int k=0; i<N-1; k++)
         if (texte[k]==mot[0] && texte[k+1] == mot[1]) nb++
      //affichage
      printf("le mot %s apparaît %d fois\n", mot, nb);
```

Question #3.4 (1 point)

Évaluer le nombre de comparaisons de votre algorithme.

Solution:

Il n'y a aucun sens à donner une complexité si on n'a pas donné d'algo !

Pour un texte de taille N, tous les mots de taille 2 sont énumérés par la boucle externe, et pour chaque mot le nombre de ses occurences sont comptées en scannant le texte entièrement. Cet algorithme coûte donc toujours $O(N^2)$ comparaisons.

Pour la deuxième solution, le nombre de comparaisons est linéaire en la taille du texte.

1 Premier programme

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

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

Compilation : dans un *terminal* :

```
gcc hello.c -o hello
```

On peut remplacer `gcc` par `clang` 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 bib. 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)
  int resu = 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 Conditionnelles

```c
  if (x%2==0){
    //instructions dans le cas "x pair"
  } else {
    //autre cas
  }
```

La deuxième branche est optionnelle.

5 Boucles for

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 points-virgules (;)
  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
}
```

6 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.

7 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. 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)!
```

8 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') {...
```

9 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); // idem, pour c un caractère (char)
scanf("%s", s); // une chaîne est un tableau, pas de &
```