Algorithmique et Programmation C (CS101) – Examen Session 1 – 2024-25

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: Le barème finalement appliqué est celui-ci :

	Exercice 1 mystère								Exercice 2 recherche par saut						Exercice 3 palindromes									
	1.1	1.2	1.3	1.4	1.5	1.6	1.7	total	2.1	2.2	2.3	2.4	2.5	total	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	total
23	1	1	1	1	1	1	1	7	1	1	1	1.5	1	5.5	1	1.5	1.5	1	1	1	1	1.5	1	10.5

et la note sur 23 a finalement constitué la note sur 20.

En rouge, mes commentaires après correction. Lorsqu'il est noté de ne pas rendre l'énoncé, il ne faut pas rendre l'énoncé. en particulier, je ne considère pas valables les annotations de sujet.

Dans les deux premiers exercices il est demandé d'écrire du C.

Les deux premier exercices s'appuieront sur les programmes ("à trous") qui, pour des questions d'efficacité de la lecture, sont placés à la dernière page du sujet.

Exercice 1 – Programme mystère

On considère le programme itératif de la Figure **??**.

Question #1.1 (1 point)

Donner sur votre copie les quelques instructions C pour déclarer et initialiser un tableau t de taille 8 avec les valeurs suivantes, dans l'ordre : $1, 2, 3, 3, 3, 5, 7, 8$ (comme si on remplissait le *main*, point <2>).

Question #1.2 ($\frac{1}{2}$ point)

Quelle instruction ajouter (au même endroit) pour réaliser l'appel de la fonction mystère avec $x = 3$, le t précédent, et la taille 8 ?

Solution:

```
int t[8]={1,2,3,3,3,5,7,8};
printf("resultat = %d\n",
mystere(3, t, 8));
```

Il n'est pas admissible de ne pas connaître la syntaxe d'initialisation d'un tableau à ce stade, surtout avec la fiche de syntaxe ! même remarque pour l'appel de fonction avec un tableau, on ne passe pas $t[8]$ en paramètre, on passe t !

Question #1.3 (1 point)

Le programme ainsi modifié réalise des impressions sur la sortie standard, lesquelles ? *Justifier en écrivant les valeurs de i, j et k à la fin de chaque corps boucle exécutée.*

Solution:

```
début :                      i=0     j=7
premiere boucle (l8) : q=3
ligne 9              : impression de "q=3"
lignes 11/12 :       t[3]==x est vrai, q=2
                     t[2]==x est vrai, q=1
                     stop car t[1]==x est faux
ligne 13             : fin, la fonction retourne 2
```

encore une fois, une "boucle if" n'a aucun sens !

Question #1.4 (1 point)

Donner une formule booléenne vraie au point de contrôle "<1>" (avant le *return*, après le *while*), en fonction de q, t, x, qui est une condition nécessaire et suffisante pour passer à ce point de contrôle.

Solution: On prend la négation de la condition de la boucle :

$$t[q] \neq x \text{ ou } q < 0$$

Question #1.5 ($1/2$ point)

Quel est la fonctionnalité de cet algorithme ?

Solution: Cet algorithme (Attention à l'orthographe de algorithme) renvoie l'indice de la premiere occurrence de x dans le tableau, s'il existe, -1 sinon. la mention d'index ou d'indice est essentielle ici ! Attention également, on travaille sur des tableaux, et non des listes.

Question #1.6 ($1/2$ point)

De quel algorithme vu en cours cet algorithme est-il une variante ? Quelle propriété doit avoir le tableau passé en paramètre ?

Solution: Il s'agit de la recherche dichotomique, qui nécessite en entrée un tableau trié croissant.

Question #1.7 (1 point)

Quelle est la complexité temporelle dans le pire des cas pour cet algorithme ? *Bien préciser "en fonction de ..."*

Solution: Une complexité qui mentionne n alors que n n'est absolument pas une entrée de la fonction mystère n'a pas de sens. La complexité temporelle au pire ici est atteinte pour un tableau d'entiers contenant $\frac{size}{2}$ fois le même élément, dans ce cas on va rechercher l'index minimal "à gauche du milieu", et pour cela on réalise $\frac{size}{2}$ fois la boucle while, pour finalement renvoyer $q = 0$. Dans ce cas, on effectue $\frac{size}{2}$ comparaisons $t[index]? = x$, la complexité en nombre de comparaisons d'éléments de tableaux, est donc au pire linéaire en la taille du tableau t.

Exercice 2 – Recherche par saut

D'après une feuille de TD de N. Meloni pour univ Toulon.

On considère un tableau trié de taille n dans l'ordre croissant. La recherche par saut utilise un paramètre fixe K et consiste à tester dans un premier temps les éléments d'indices K, K, $2K$ etc., pour déterminer un "sous tableau", puis, dans un deuxième temps à faire une recherche séquentielle dans ce sous-tableau. Par exemple avec $T = [1,2,3,4,5,6,7,8,9,10]$ et $K = 3$, pour rechercher 6 (et renvoyer l'index 5), l'algorithme va tester successivement les nombres $1,4,7,5,6$, en déduire que le nombre cible est dans l'intervalle d'index de tableau entre 3 et 6 si il existe, et enfin va parcourir le sous tableau correspondant. Dans la suite, nous allons remplir le programme de la Figure **??**.

Question #2.1 (1 point)

Compléter la spécification de la fonction `cherche_indice`. Sur l'exemple ci-dessus, que doit renvoyer cette fonction ?

> **Solution:** Calcule et renvoie i, multiple de K, tel que : si l'élément est dans le tableau, alors il est forcément dans l'intervalle entier $[i, K+i[$. La fonction renvoie -1 si l'élément n'est pas dans le tableau. Dans le cas présent, la fonction renvoie 3.Faites attention au découpage des questions, ici l'exercice découpait le problème en 2, et en deux fonctions. J'ai distribué des points entre les 2 questions aux personnes qui ont directement traité le problème dans la question 2.2.

Question #2.2 (1 point)

Implémenter le corps de la fonction `cherche_indice` (point "<3>"). *Merci de bien expliquer ce que vous faites sur votre copie.*

> **Solution:**
>
> ```
> int cherche_indice(int t[], int size, int el){
>   int i=0;
>   if (t[0] > el || t[size-1] <el) return -1;
>   else
>     while (i<size-K){
>       printf("i=%d,\n", i);
>       if (t[i+K]>el) return i;
>       i = i+K;
>     }
>   return i;
> }
> ```
>
> Ainsi, on parcourt le tableau en ne regardant que les cases d'index multiples de K, à partir de 0, en faisant attention à ne pas dépasser. Dès que la case observée dépasse l'élément cherché, on sait qu'on a trouvé le bon intervalle.

Question #2.3 (1 point)

Au point de contrôle "<4>" (dans les accolades), donner un invariant le plus précis possible (en fonction de t, $size$, i, et el).

> **Solution:** Sans surprise, c'est une conséquence de la spécification de la fonction précédente. L'élément existe dans le tableau, quelque part entre les index i et $i+K$, ce qui peut s'écrire comme ceci :
>
> $$\exists j, i \leq j < i+K, t[j] = el$$

Question #2.4 (1 point)

Terminer l'implémentation de la fonction `cherche_element_index` (point "<4>") et du main (<5>)

Solution:

```
    if (i!= -1){ //<4>
      for (int j=i; j<i+K; j++){
          if (t[j]==el) return j;// trouvé
    }
```

On réalise une recherche linéaire dans le sous tableau entre i et $i+K$. et pour le main :

```
    //<5>
  i = cherche_element_index(t, 10, 6);
  if (i!=-1) {
    printf("found! index=%d\n",i);
  }
```

Question #2.5 (1 point)

Étudier le nombre de comparaisons au pire, en fonction de K et n (à une constante près).

Solution: Le pire cas est lorsque l'élément est à la toute droite du tableau, car : le cherche indice réalise alors les comparaisons pour K, $2K$, ... (c'est à dire en gros $\frac{n}{K}$ comparaisons (avec n la taille du tableau passé en paramètre, cad "size"). Ensuite, la recherche de l'index effectue K comparaisons. Cela fait $\frac{n}{K}+K$ comparaisons au pire. Il aurait été plus judicieux de remplacer n par size dans l'énoncé de cette fonction. idem k versus K

Exercice 3 – Comptons des palindromes

D'après un (début de) sujet de CCP-INP 2023, typographié par François Nath, et adapté d'après le corrigé par Lilian Besson, Aude Le Gluher.

En langue française, "ressasser" est le mot palindrome le plus long, tandis qu'il semble que "saippuakauppias" (désignant un marchand de savon en Finlande) soit le plus long mot palindrome au monde. L'objet de cette partie est de compter le nombre de palindromes présents dans un mot donné.

Dans la suite on considère des mots (notés u, v, w) de n lettres. On nomme ε le mot vide (de taille 0). La longueur d'un mot u (ou sa taille) est le nombre de lettres. Les indices des lettres commencent à 0, comme en C, et enfin, on notera indifféremment u_i et $u[i]$.

Définition 1 (Miroir)
Soit u un mot de taille n. Le *miroir* de $u = u_0 \dots u_{n-1}$, noté $\overline{u}$, est le mot $\overline{u} = u_{n-1} \dots u_0$. Par convention $\overline{\varepsilon} = \varepsilon$. Par exemple, le mot miroir de `tagada` est `adagat`.

Définition 2 (Palindrome)
u est un *palindrome* si et seulement si $u = \overline{u}$. Par convention, le mot vide ε est un palindrome.

On recherche donc dans cette partie le nombre de (sous-) palindromes contenus dans u (**qui ne sont pas vides**.

Les algorithmes fournis sont en pseudo-code, il est demandé ici d'écrire du pseudo code "qui ressemble".

Question #3.1 (1 point)
Montrer que le mot $u = babb$ possède 6 sous-palindromes non vides, en les exhibant avec leur multiplicité.

Solution: Les palindromes contenus dans u sont les mots suivants : b (trois fois), a (une fois), bb et bab chacun une fois, soit **6 sous-palindromes**
Le mot palindrome est écrit dans l'énoncé, merci d'utiliser la même orthographe.

L'Algorithme1 en pseudo-code propose de parcourir naïvement la chaîne u, afin de compter son nombre de palindromes.

Algorithm 1: Premier algorithme pour compter les palindromes

```
Fonction comptenaif(u,n) :Entier
    D: u mot de taille n
    L: i,j,k :Entiers
    nb ← 0 ;
    Pour i de 0 à n-1 Faire
        Pour j de i+1 à n Faire
            estPalindrome ← True;
            Pour k de 0 à j-i-1 Faire
                Si u[i+k] ≠ u[j-1-k] alors
                    estPalindrome ← False;
                    sortir du Pour k
            Si estPalindrome alors
                nb ← nb + 1
    Retourner nb
```

Question #3.2 (1 point)
Représenter dans un tableau les étapes effectuées pour $i = 0$ et $j = 3$ pour le mot u de la question précédente. Quels éventuels palindromes avons nous comptés ?

Solution: On fixe donc $i = 0$ et $j = 3$. Soit n_0 la valeur du nombre avant cette étape. La boucle for sur k ira donc de 0 à 2 compris.

```
k=0      u[0] ?= u[2] oui, on continue
k=1      u[1] ?= u[1] oui, on continue
k=2      u[2] ?= u[0] oui
fin de la boucle sur k,
```

On augmente le nombre de palindromes de 1 car la variable booléenne n'a pas été modifiée. Le palindrome trouvé ici est $u[0..3[$ c'est-à-dire *bab*.

Question #3.3 (1½ points)

Évaluer la complexité dans le pire des cas de l'Algorithme 1 en fonction de n.

Solution: L'algorithme 1 est constitué soit de boucle **for** (sur i en n, sur j en n, et sur k en $j - i$), soit de lignes qui sont en temps constants (des affectations, des tests, des calculs arithmétiques). Sa complexité temporelle peut donc être bornée (dans le pire des cas, où aucun des "Sortir du pour k" n'est appelé) par $\mathcal{O}(1) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=i}^{j-1}\mathcal{O}(1)$. La somme se borne par $\mathcal{O}(n^3)$, et donc la complexité au pire cas de l'algorithme 1 est $\mathcal{O}(n^3)$ (cubique).

On souhaite bien sûr améliorer cette première idée. Pour ce faire, on utilise tout d'abord le paradigme de la programmation dynamique (que vous verrez plus tard dans le cursus, ici, on utilise juste un exemple). On considère toujours un mot u de taille n. On note $u[i, j]$ le sous-mot de u entre les index i et $j - 1$ compris.

Question #3.4 (1 point)

Soit $u[i, j]$ un sous-mot de u, avec $0 \leq i, j \leq n$ et $i + 2 \leq j$. À quelles conditions sur u_i, u_{j-1} et $u[i + 1, j - 1]$ le mot $u[i, j]$ est-il un palindrome ? Pour expliquer, vous prendrez le mot $u = babb$, de taille 4, et le sous mot $w = bab = u[0, 3]$.

Solution: $u[i, j] = u_i \dots u_{j-1}$ est un palindrome ssi $u_i u_{i+1} \dots u_{j-2} u_{j-1} = u_{j-1} u_{j-2} \dots u_{i+1} u_i$ ssi $u_i = u_{j-1}$ et $u_{i+1} \dots u_{j-2} = u_{j-2} \dots u_{i+1}$ ssi $u_i = u_{j-1}$ et $u[i + 1, j - 1]$ est un palindrome. Si vous voulez quantifier, alors il ne faut prendre ni j, ni i, ici ils sont fixés.

En utilisant la relation précédente, on va construire un tableau P de booléens de taille $(n + 1) \times (n + 1)$, $P[i][j]$ étant vrai si $u[i, j]$ est un palindrome. Ensuite, il s'agira de compter le nombre de cases à vrai dans le tableau.

Question #3.5 (½ point)

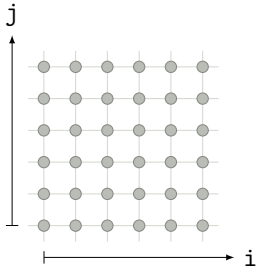
Expliquer pourquoi, pour tout $i \in [\![0, n - 1]\!]$, $P[i][i + 1] =$ `Vrai`.

Solution: Ce sont des mots de taille 1, donc des palindromes.

D'après la question 3.4, pour tout $0 \leq i, j \leq n, i + 2 \leq j$: $P[i][j] = \big((u[i] == u[j - 1])$ `et` $P[i + 1][j - 1]\big)$

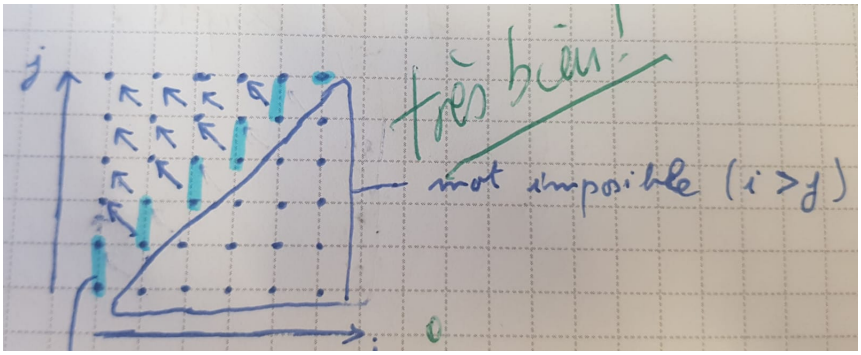
Question #3.6 (1 point)

Reproduire sur votre copie le schéma suivant de la matrice P avec des flèches décrivant l'ordre imposé des calculs ($P_{i,j} \to P_{i',j'}$ dénote $P_{i,j}$ doit être calculé avant $P_{i',j'}$).



Solution: Il faut dessiner des flèches $(i,j) \to (i-1, j+1)$ Attention, les points $(i, i+1)$ ne doivent pas etre destinataires des flèches car ces points n'ont pas de dépendance de calcul. Les points pour $j > i+2$ n'ont pas de sens. Les points sur la diagonale correspondent aux mots vides. Les points au dessus de la diagonale correspondent aux mots de taille 1. Les autres sont calculés par la relation de récurrence citée.

Dessin d'une copie :



On fournit maintenant le début du pseudo-code d'un deuxième algorithme (Algorithme 2), basé sur les relations de récurrence précédentes.

Algorithm 2: Deuxième algorithme pour compter les palindromes

```
Fonction compte2(u,n) :Entier
    D: u mot de taille n
    L: i, j : Entiers
    L: matrice P de (n+1) × (n+1) Booléens, initialisés à Faux.
    Pour i de 0 à n Faire
        P[i][i] ← Vrai
    Pour i de 0 à n-1 Faire
        P[i][i+1] ← Vrai
    Pour i <à remplir 1> Faire
        Pour j <à remplir 2> Faire
            P[i][j] ← (u[i] == u[j-1]) et P[i+1][j-1]
    nb ← 0;
    <à remplir 3>;
    Retourner nb
```

Question #3.7 (1 point)

En faisant attention à l'ordre de remplissage de la matrice P et aux indices, compléter les deux boucles "Pour" (à remplir 1, et à remplir 2).

Solution:

```
Pour i = n-1 a 0 (decroissant) Faire :
```

```
    Pour j = i+2  à n Faire :
```

Question #3.8 (1½ points)

Terminer la fonction "à remplir 3", pour finalement compter le nombre de palindromes de u. **On n'oubliera pas de "décompter" le mot vide.**

Solution:

```
Pour i = 0 a n-1 Faire :
    Pour j = 0 a n-1 Faire :
        Si P[i][j] Alors :
            nombre_palindromes += 1
    Fin Pour
Fin Pour
nombre_palindromes -= (n+1)// il faut supprimer toute la diagonale
```

Question #3.9 (1 point)

Quelle est la complexité en temps et en mémoire, dans tous les cas ?

Solution: Cet algorithme de programmation dynamique utilise une mémoire quadratique en $\mathcal{O}(n^2)$ (la matrice P), et prend ce même temps pour l'initialiser puis pour la remplir, et enfin ce même temps pour la parcourir et calculer le nombre voulu ;

Solution: Le lecteur ou la lectrice curieux.se pourra se reporter au sujet CCP INP 2023 pour y décrouvrir un algo linéaire en temps et en mémoire pour le même problème.