Cours d'introduction à l'informatique
Préparation à l'examen
Correctif

1. (a) Une procédure possible consiste à récupérer la longueur de la chaîne de caractères puis d'échanger itérativement les éléments à l'indice *i* du tableau avec ceux en *longueur - i -1*. Comme nous avons un pointeur donné en entrée, les éléments seront directement modifiés et pas seulement localement à la procédure. L'algorithme peut alors s'écrire :

```
void inversion(char* c){
    unsigned longueur;
    for(longueur = 0; *(c+longueur); longueur++);
    for(unsigned i = 0; i < longueur/2; i++){
        char tmp = c[longueur-i-1];
        c[longueur-i-1] = c[i];
        c[i] = tmp;
    }
}
```

   (b) La complexité en temps de cette procédure est $\mathcal{O}(n) + \mathcal{O}(n/2)$, où n correspond à la taille de la chaîne de caractères. Cette complexité est donc approximée par $\mathcal{O}(n)$, où n correspond à la taille de la chaîne de caractères. Pour obtenir ce résultat, nous avons calculé la complexité de la première boucle qui va de 0 jusqu'à la *taille de la chaîne de caractères*, et celle de la deuxième boucle, allant de 0 jusqu'à *longueur/2*. Toutes les opérations à l'intérieur de ces boucles peuvent être effectuées en temps constant. On obtient donc bien $n + n/2$. La complexité théorique correspond à la plus grande complexité, qui est donc bien $\mathcal{O}(n)$.

2. (a) Une fonction possible consiste à parcourir le tableau itérativement. Chaque fois qu'un nouvel élément apparait, la taille est augmentée de 1 et on ajoute cette valeur à l'endroit correspondant à l'ancienne taille. Pour vérifier qu'un élément est différent, on peut simplement regarder que le dernier élément ajouté est différent de celui qu'on regarde. Ceci est possible parce que le tableau est trié. On est donc

sûr qu'on ne peut voir que la valeur du dernier élément ajouté ou une valeur supérieure à ce dernier. L'algorithme peut s'écrire :

```
unsigned doublon(int* v, unsigned n){
    unsigned nouvelle_taille = 1;
    for(unsigned i = 1; i < n; i++)
        if(v[nouvelle_taille-1] != v[i])
            v[nouvelle_taille++] = v[i];
    return nouvelle_taille;
}
```

(b) On souhaite établir la validité du triplet suivant :

$\{n > 0, v = [v_0, v_1, \ldots, v_n{-}1]$ où $\forall j \in [0, n{-}1] : v_j \leq v_{j+1}, \text{nouvelle\_taille} = 1\}$

```
for(unsigned i = 1; i < n; i++){
    if (v[nouvelle_taille-1] != v[i])
        v[nouvelle_taille++] = v[i];
}
```

$\{\text{nouvelle\_taille} = \text{taille du tableau ne contenant que des entiers qui ne sont pas des doublons, } v = [v_0, v_1, \ldots, v_{\text{nouvelle\_taille}-1}, \ldots, v_{n-1}], \text{ où aucun élément n'est répété dans } v_0, \ldots, v_{\text{nouvelle\_taille}-1}\}$

En décomposant la boucle *for*, on obtient le triplet équivalent :

$\{n > 0, v = [v_0, v_1, \ldots, v_n{-}1]$ où $\forall j \in [0, n{-}1] : v_j \leq v_{j+1}, \text{nouvelle\_taille} = 1, i = 1\}$

```
while(i < n){
    if (v[nouvelle_taille-1] != v[i])
        v[nouvelle_taille++] = v[i];
    i++;
}
```

$\{\text{nouvelle\_taille} = \text{taille du tableau ne contenant que des entiers qui ne sont pas des doublons, } v = [v_0, v_1, \ldots, v_{\text{nouvelle\_taille}-1}, \ldots, v_{n-1}], \text{ où aucun élément n'est répété dans } v_0, \ldots, v_{\text{nouvelle\_taille}-1}\}$

Pour trouver un invariant de boucle I, on caractérise les opérations effectuées par la boucle jusqu'à une itération donnée. Un invariant

possible est :

I :$n > 0$
et $1 \leq i \leq n$
et nouvelle_taille $=$ nombre d'entiers différents entre v_0 et v_{i-1}
et $1 \leq$ nouvelle_taille $\leq i$
et $v_0, \ldots, v_{\text{nouvelle\_taille}-1}$ n'a pas de doublon

Cet invariant exprime qu'avant et après chaque itération de la boucle, la valeur de nouvelle_taille correspond au nombre d'entiers différents entre v_0 et v_{i-1}, et il n'y a aucun doublon entre v_0 et $v_{\text{nouvelle\_taille}-1}$. Montrons maintenant que cet invariant est valide :

- Initialement, on a nouvelle_taille = 1, et $i = 1$, on a bien qu'il n'y a pas de doublon dans un vecteur composé d'un seul entier (entre v_0 et v_{i-1}) et que le nombre d'éléments différents d'un vecteur d'un seul entier est 1. L'invariant est satisfait.
- Pour chaque itération de la boucle, on a le triplet :

$$\{I, i < n\}$$

```
if (v[nouvelle_taille-1] != v[i])
    v[nouvelle_taille++] = v[i];
i++;
```

$$\{I\}$$

Montrons que ce triplet est valide en notant respectivement x et x' la valeur d'une variable x avant et après l'itération concerncée.

- On a $i' = i + 1$ dans tous les cas, ce qui implique que $1 \leq i' \leq n$.
- Il y a 2 cas à considérer :
  i. Si $v_{\text{nouvelle\_taille}-1} \neq v_i$, alors v_i n'est pas déjà entre v_0 et $v_{\text{nouvelle\_taille}-1}$ et il est placé dans $v_{\text{nouvelle\_taille}}$ avant d'avoir nouvelle_taille' = nouvelle_taille+1.
  On obtient donc bien que

$$\begin{aligned} v' &= v_0, \ldots, v_{\text{nouvelle\_taille}}, \ldots v_i, \ldots v_{n-1}, \\ &\text{où } v_0, \ldots, v_{\text{nouvelle\_taille}} \text{ ne contient pas de doublon} \\ &= v_0, \ldots, v_{\text{nouvelle\_taille'}-1}, \ldots v_{i'-1} \ldots v_{n-1} \end{aligned}$$

  ii. Sinon, ni v ni nouvelle_taille ne sont modifiés on obtient donc bien que

$$\begin{aligned} v' &= v_0, \ldots, v_{\text{nouvelle\_taille}-1}, \ldots v_i, \ldots v_{n-1}, \\ &\text{où } v_0, \ldots, v_{\text{nouvelle\_taille}-1} \text{ ne contient pas de doublon} \\ &= v_0, \ldots, v_{\text{nouvelle\_taille'}-1}, \ldots v_{i'-1} \ldots v_{n-1} \end{aligned}$$

- En fin de boucle, on a $\{I,\ i \geq n\}$, ce qui implique que $i = n$ et donc que nouvelle_taille contient le nombre d'entiers différents entre v_0 et v_{n-1}, et qu'il n'y a aucun doublon entre v_0 et $v_{\text{nouvelle\_taille}-1}$.

Il reste à démontrer que la boucle se termine. Il suffit pour cela de considérer le variant de boucle $v = n - i$ On a $1 \leq i \leq n$, ce qui entraîne que le variant possède toujours une valeur entière non négative. De plus, chaque itération de la boucle diminue la valeur du variant.

3. (a) Cette fonction calcule la somme des chiffres d'un nombre entier. Pour facilement s'en apercevoir, on peut simuler une exécution. Regardons ce que fait ce code quand on lui donne en entrée le nombre 1504 :
   - $f(1504) = 4 + f(150)$
   - $f(150) = 0 + f(15)$
   - $f(15) = 5 + f(1)$
   - $f(1) = 1 + f(0)$
   - $f(0) = 0$ (cas de base)

   Ceci correspond à l'empilement généré par les appels récursifs. Dépilons la pile pour trouver notre résultat :
   - $f(0) = 0$
   - $f(1) = 1 + 0$
   - $f(15) = 5 + 1$
   - $f(150) = 0 + 6$
   - $f(1504) = 4 + 6 = 10$

   (b) Dans le pire des cas, un appel à la fonction pour traiter un entier > 0 conduit à un appel récursif de cette même fonction pour traiter un entier $n/10$, et ainsi de suite jusqu'à arriver au cas de base $n = 0$. La profondeur de récursion est donc égale à $\mathcal{O}(\log_{10} n+1)$. La complexité en espace est donc $\mathcal{O}(\log\ n)$. Regarder à notre simulation précédente peut aider à comprendre ceci.

   (c) Une fonction possible est :

```
int f(int n){
    int resultat;
    for(resultat = 0; n > 0; n/=10)
        resultat += n%10;
    return resultat;
}
```

4. (a) Un type structuré possible est :

```
typedef struct {
    unsigned elements[100];
    unsigned nb_elements;
} pile;
```

(b) Une fonction possible consiste à vérifier que la pile n'est pas pleine avant d'ajouter la valeur n à l'emplacement d'indice donné par *nb_elements* qui est stocké dans la structure. Il ne faut pas oublier de mettre cette valeur à jour après. Cet algorithme peut s'écrire :

```
unsigned empile(pile* p, unsigned n){
    if (p->nb_elements == 100)
        return 0;
    p->elements[p->nb_elements++] = n;
    return 1;
}
```

(c) Une fonction possible consiste à d'abord vérifier que la pile n'est pas vide avant de simplement modifier *nb_elements*. L'algorithme peut s'écrire :

```
int depile(pile* p){
    if (p->nb_elements == 0)
        return -1;
    return p->elements[p->nb_elements--];
}
```

5. (a) Un type structuré possible est :

```
struct elem_t{
    int valeur;
    struct elem_t* precedent;
    struct elem_t* suivant;
};
typedef struct elem_t elem;
```

(b) Une fonction possible consiste à trouver le dernier élément en bouclant sur l'élément suivant de e, avant d'allouer un nouvel élément de lui donner sa valeur et de bien le mettre comme élément suivant de e et de mettre e comme son élément précédent. L'algorithme peut s'écrire :

```
elem *ajout(elem* e, int n){
    while(e->suivant)
        e = e->suivant;
    elem *new_e = malloc(sizeof(elem));
    if (!new_e)
        return NULL;
    e->suivant = new_e;
    new_e->precedent = e;
    new_e->suivant = NULL;
    new_e->valeur = n;
    return new_e;
}
```

(c) Une fonction possible consiste à boucler sur l'élément suivant tout en additionnant sa valeur au résultat final à chaque itération. L'algorithme peut s'écrire :

```
int somme(elem* e){
    int resultat;
    for (resultat = 0; e; e=e->suivant)
        resultat += e->valeur;
    return resultat;
}
```

(d) Une fonction possible consiste à boucler sur l'élément précédent tout en multipliant sa valeur au résultat final à chaque itération. L'algorithme peut s'écrire :

```
int produit(elem* e){
    int resultat;
    for (resultat = 1; e; e=e->precedent)
        resultat *= e->valeur;
    return resultat;
}
```

(e) Une fonction possible consiste à chaque fois récupérer l'élément suivant grâce à une variable temporaire, avant de libérer l'espace de l'élément actuel. L'algorithme peut s'écrire :

```
void liberer(elem* e){
    elem *p;
    for(p = e; p; e = p){
        p = e->suivant;
        free(e);
    }
}
```