TD n°6 Corrigé

Exercice 1

Pour la correction on ne comptera que les printf, de toute façon compter les printf, les additions ou les comparaisons donne le même résultat. (il faut toujours soit compter l'opération qui est faite le plus de fois, soit tout compter)

Pour $f1$ la première boucle for fait n fois un affichage, donc coute n opérations. La deuxième boucle for fait la même chose. On a donc $2n$ affichages et la complexité asymptotique est en $O(n)$.

Pour $f2$ si on somme les opérations des boucles on trouve qu'il y a $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2$ affichages, soit $2n^2$ en simplifiant. La complexité est un $O(n^2)$.

Pour $f3$ si on somme les opérations des boucles on trouve qu'il y a $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 2$ affichages, soit $\sum_{i=0}^{n-1} 2(n-1-i) = 2n^2 - 2n - (n-1)n = n^2 - n$. La complexité est un $O(n^2)$.

Pour $f4$ si on somme les opérations des boucles on trouve qu'il y a $\sum_{i=2}^{n+3}\sum_{j=i-2}^{i+2} 2$ affichages, soit $\sum_{i=2}^{n+3} 5 * 2 = 10 * (n-4)$.

La complexité est un $O(n)$.

Exercice 2

1. Pour trouver les doublons, on peut trier et chercher les doublons dans un tableau trié, ce qui coute un $O(n\log(n))$ plus un $O(n)$. Or un $O(n)$ est un $O(n\log(n))$ (n est dominé par $n\log(n)$), donc cette méthode coûte au total un $O(n\log(n))$.
   L'autre méthode consiste à directement chercher les doublons, ce qui nous coûte un $O(n^2)$ opérations. Ainsi il est plus efficace de trier et de recherche dans le tableau trié.

2.
```c
bool recherche_trie(int* tab, int n){
  for(int i = 1;i<n;i+=1){
    if (tab[i]==tab[i-1]){return true;}
  }
  return false;
}
```

3.
```c
bool recherche_pas_trie(int* tab, int n){
  for(int i = 0;i<n;i+=1){
    for(int j = 0;j<n;j+=1){
      if (i!=j && tab[i] == tab[j]){return true;}
  }
  return false;
}
```

4. Terminaison : on ne fait que des boucles for.
   Correction :
   Pour `recherche_trie` un invariant est "à la fin de l'itération i, si aucun retour n'a été fait, il n'y a pas de doublons dans le sous-tableau de **`tab`** pour les indices entre 0 et i".
   Pour `recherche_pas_trie` un invariant de la boucle interne, à i fixé, est "à la fin de l'itération j, si aucun retour n'a été fait, les éléments du tableau situés aux indices entre 0 et j sont différents de **`tab[i]`**".
   Un invariant de la boucle externe est : "à la fin de l'itération i, si aucun retour n'a été fait, les éléments du sous-tableau pour les indices entre 0 et i ne sont présents qu'une seule fois dans le tableau entier".
   Il n'y a plus qu'à rédiger la preuve qu'il s'agit bien d'invariants et qu'ils permettent d'exprimer la correction à la fin du programme.

Exercice 3

Pour les complexités, on donnera un ordre de grandeur, avec les notations de Landau, et non pas un décompte précis des opérations élémentaires.

1. Écrire une fonction C `bool egal(int* tab1, int* tab2, int n1, int n2)` qui vérifie si deux tableaux sont égaux, c'est à dire qu'ils ont même taille et même contenu. Donner sa complexité dans le meilleur et dans le pire cas, en temps et en espace.

```c
bool egal(int* tab1, int* tab2, int n1, int n2){
  if (n1!= n2){return false;}
  else{
    for(int i=0; i<n1;i+=1){
```

```
      if(tab1[i]!=tab2[i]){return false;}
    }
    return true;
  }
}
```

Dans le meilleur cas, $n_1 \neq n_2$ et on entre pas dans la boucle. La complexité est alors $O(1)$.

Dans le pire cas les tableaux sont égaux et la boucle s'arrête car i atteint $n_1 - 1$. La complexité est alors $O(n)$ puisqu'on fait n itérations et 2 ou 3 opérations élémentaires à chaque itération.

La complexité temporelle est en $O(1)$, on a juste créé une variable i entière.

2. Écrire en C une fonction `int minimum(int* tab, int n)` qui renvoie le plus petit élément d'un tableau. Donner sa complexité dans le meilleur et dans le pire cas, en temps et en espace.

```
int minimum(int* tab, int n){
  int res = tab[0];
  for(int i=1; i<n;i+=1){
    if(res<tab[i]){res = tab[i];}
  }
  return res;
}
```

Ici la fonction fait toujours $n-1$ itérations de la boucle, il n'y a pas de cas meilleur qu'un autre. La complexité est toujours en $O(n)$, on fait $n-1$ itérations d'une boucle qui effectue un nombre constant d'opérations.

La complexité mémoire est en $O(1)$, on a juste créé deux variables, i et res.

3. Écrire en C une fonction `bool recherche(int* tab, int n, int x)` qui cherche un élément `x` dans un tableau `tab`. Donner sa complexité dans le meilleur et dans le pire cas, en temps et en espace.

```
bool recherche(int* tab, int n, int x){
  int i = 0;
  while(i<n){
    if(tab[i] == x){return true;}
    i+=1;
  }
  return false;
}
```

Le meilleur cas est lorsque x se trouve dans la case 0 du tableau, on effectue un seul tour de boucle et $O(1)$ opérations.

Le pire cas est quand l'élément ne se trouve pas dans le tableau, on est obligés de parcourir tout le tableau pour s'en rendre compte. La complexité est alors en $O(n)$.

La complexité spatiale, à nouveau, est $O(1)$.

Exercice 4

Calculer proprement la complexité des fonction récursives suivantes :

```
int f1(int n){
  if (n==0){return 1;}
  else{return 1+f1(n/2);}
}
```

```
int f2(int n){
  if (n==0||n==1){return n;}
  else {return f2(n-1);}
}
```

```
bool f3(int n, int m){
  if (m==1) {return false;}
  else if (n==1){return true;}
  else if (n>m){return f3(n-1,m);}
  else {return f3(n,m-1);}
}
```

```
int f4(int n){
  if (n==0 || n==1){return n;}
  else {return f4(n-1)+f4(n-2);}
}
```

On commence par donner les formules de récurrence pour la complexité de chacune de ces fonctions. On notera C_i la complexité de la fonction `fi`. :

- $C_1(0) = 1$ et $C_1(n) = 3 + C_1(\lfloor n/2 \rfloor)$ (on compte un test, un + et un /)
- $C_2(0) = 1$ et $C_2(1) = 1$ et $C_2(n) = 5 + C_1(n-1)$ (on compte deux tests et trois opérations arithmétiques)
- $\forall m \in \mathbb{N}, C_3(1,m) = 1$ et $\forall n \in \mathbb{N}, C_3(n,1) = 1$ et si $n > m$ alors $C_3(n,m) = C_3(n-1,m) + 1$ et sinon $C_3(n,m) = C_3(n,m-1) + 1$ (ici j'ai choisis une constante de 1 parce que les constantes additives ne sont pas importantes)
- $C_4(0) = 1$ et $C_4(1) = 1$ et $C_4(n) = C_4(n-1) + C_4(n-2) + 5$.

On va maintenant procéder à leur étude.

Pour f1 :

On suppose que $n = 2^k$ est une puissance de 2 (avec $k \geq 0$).

En itérant la formule de récurrence pour remplacer $C_1(n/2)$ par $C_1(n/4)$ puis par $C_1(n/8)$ etc..., on voit la formule suivante apparaitre, que l'on démontrera par récurrence :

Pour $0 \leq l \leq k$, $\mathcal{P}_l : C_1(n) = C_1(n/2^l) + l * 3$

On initialise pour $l = 0$, on doit montrer $C_1(n) = C_1(n)$ ce qui est trivial.

Supposons maintenant $C_1(n) = C_1(n/2^l) + 3l$, alors en utilisant la formule de récurrence de C_1 on a $C_1(n) = C_1(n/2^{l+1}) + l + 3l = C_1(n/2^{l+1}) + 3(l+1)$.

La formule est donc vraie par principe de récurrence pour $0 \leq l \leq k$. En particulier elle est vraie en $l = k$. On a $C_1(n) = C_1(n/2^k) + 3k = C(1) + 3k = C(0) + 3 + 3k = 4 + 3k = 4 + 3\log_2(n)$.

On en conclut que la complexité est en $O(\log_2(n))$.

Pour f2 :

La suite $(C_2(n))_{n\in\mathbb{N}}$ est arithmétique à partir du rang 1. On a donc pour tout $n \geq 1, C_2(n) = 5(n-1)+1$ et $C_2(0) = 1$. La complexité est en $O(n)$.

Pour f3 :

Supposons initialement $m \geq n$. Alors m va diminuer de 1 en 1 jusqu'à atteindre 1 et n ne va pas changer. La complexité est donc en $O(m)$.

Supposons maintenant $n > m$. Alors n diminue de 1 et m ne change pas jusqu'à ce qu'on revienne dans le cas précédent. Cela prend $n - m$ étapes pour que n redevienne plus petit que m. Ensuite cela prend $m - 1$ étapes pour que m arrive à 1. Au total dans la relation de récurrence il y aura $n - m + m - 1$ appels, chacun ajoutant 1 à la valeur (et on part de 1). Au total c'est un $O(n)$.

Pour unifier les deux cas, on peut dire que la complexité est en $O(max(n, m))$.

Pour f4 : (à lire quand vous saurez trouver le terme général de suites récurrentes d'ordre 2)

On pose $A_n = C(n) + 5$, A_n vérifie la relation de récurrence $A_n = A_{n-1} + A_{n-2}$.

L'équation caractéristique de cette récurrence d'ordre 2 est $r^2 - r - 1 = 0$, dont on cherche les solutions. On réécrit l'équation en $(r - 1/2)^2 - 1/4 - 1 = 0$.

On doit donc résoudre $(r - 1/2)^2 = 5/4$, soit $r = \dfrac{\sqrt{5}+1}{2}$ ou $r = \dfrac{-\sqrt{5}+1}{2}$. On note φ la première racine, l'autre est $-1/\varphi$.

L'expression de la suite est donc de la forme $A_n = A\varphi^n - B/\varphi^n$. En utilisant les conditions initiales $A_0 = 6$ et $A_1 = 6$, on obtient $A = 9 - 3\sqrt{5}$ et $B = 3 - 3\sqrt{5}$.

Au final on a $C_4(n) = (9 - 3\sqrt{5})\varphi^n - (3 - 3\sqrt{5})/\varphi^n - 5$. En notant que $|\varphi| > 1$, on déduit que le premier terme est un $O(\varphi^n)$ et le deuxième aussi. La complexité finale est exponentielle.