TD : COMPLEXITÉ

Exercice 1 : premiers calculs

```
RECHERCHE (T matrice carrée d'entiers, e entier) :
    res ← FAUX
    n ← TAILLE(T)
    i ← 0
    TANT QUE i < n ET res = FAUX FAIRE
        j ← 0
        TANT QUE j < n ET T_{i,j} ≠ e FAIRE
            j ← j + 1
        FIN TANT QUE
        SI j ≠ n ALORS
            res ← VRAI
        FIN SI
        i ← i + 1
    FIN TANT QUE
    RENVOYER res
```

On appelle $C_+(n)$ le nombre d'additions effectuées par cet algorithme `RECHERCHE` pour une matrice de taille $n \times n$. On suppose que le nombre d'additions effectuées par l'appel à la fonction `TAILLE` est de 0. Il nous reste à étudier les boucles.

- Pour la boucle interne, on voit qu'il n'y a qu'une addition effectuée dans le corps de la boucle. Le nombre d'itérations ne peut pas être calculé précisément car la boucle est non bornée avec deux conditions de sortie. On va donc devoir utiliser un encadrement. Au minimum, la boucle peut ne faire aucune itération (si $T_{i,j}$ `= e`). Au maximum, la boucle peut faire autant d'itérations que nécessaires pour que `j` atteigne `n`. On obtient donc un nombre d'additions effectuées par cette boucle compris entre 0 et $\sum_{j=0}^{n-1} 1$.
- Pour la boucle externe, le nombre d'additions effectuées dans le corps de la boucle est de 1 + le nombre d'additions effectuées par la boucle interne. Pour le nombre d'itérations, on doit à nouveau utiliser un encadrement car la boucle est non bornée avec deux conditions de sortie. Au minimum, on fera 1 itération (si `res` devient `VRAI` dès le premier tour). Au maximum, on fera autant d'itérations que nécessaires pour que `i` atteigne `n`.

On obtient alors l'encadrement suivant pour le nombre d'additions de notre fonction :

$$1 \times (1+0) \leqslant C_+(n) \leqslant \sum_{i=0}^{n-1} \left(1 + \sum_{j=0}^{n-1} 1\right)$$

1. Montrez donc que $1 \leqslant C_+(n) \leqslant n(n+1)$.

Une fois l'encadrement obtenu, on peut alors exprimer notre complexité avec les notations de Landau : $C_+(n) = \Omega(1)$ et $C_+(n) = \mathcal{O}(n^2)$. Intuitivement, on voit bien que dans le meilleur des cas, on trouve l'élément dans la première case et on s'arrête tout de suite, alors que dans le pire des cas la valeur n'est pas présente et on doit parcourir toute la matrice. Mais ce n'est absolument pas une preuve ! Quand on vous demande une complexité dans le meilleur ou le pire des cas, il faut **impérativement la calculer proprement** comme nous l'avons fait ci-dessus. Simplement dire, "le pire des cas est …. et sa complexité est de ….." n'est **absolument pas** une justification correcte. Vous ne faîtes que calculer un cas particulier, sans jamais justifier que ce cas est bien le pire. **Donner un cas particulier n'est dont pas une justification acceptable pour un meilleur ou un pire des cas.**

2. Calculez maintenant $C_{\text{comp}}(n)$, le nombre de comparaisons avec `e` effectuées par cet algorithme (ce nombre vaut 0 pour la fonction `TAILLE`).

3. Calculez aussi $C_{\text{tout}}(n)$, qui compte toutes les opérations élémentaires effectuées par l'algorithme (additions, comparaisons, affectations, opérations booléennes, accès à un élément du tableau, appel à `TAILLE`).
4. Comparez les trois complexités obtenues.

Lorsqu'une des opérations de l'algorithme est plus significative que les autres, on s'intéresse à cette opération dans le calcul de la complexité. Mais souvent, sans plus de précision du sujet, on considère que toutes les opérations élémentaires ont un coût similaire et on exprime une complexité "générale", où toutes les opérations comptées sont en $\mathcal{O}(1)$.

5. Un encadrement n'est pas toujours nécessaire pour la complexité, notamment si la boucle est bornée. Modifiez l'algorithme précédent pour utiliser deux boucles `POUR`. Calculez alors *proprement* la complexité.

Dans le cas des fonctions récursives, on obtient naturellement une relation de récurrence pour la complexité. Il s'agit ensuite de trouver le terme général.

```
FONCTION F(L liste non vide d'entiers, e entier)
    SI est_vide(queue(L)) ALORS
        RENVOYER e × tête(L)
    SINON
        RENVOYER e × tête(L) + F(queue(L), e+1)
    FIN SI
FIN FONCTION
```

On note C_n la complexité de la fonction `F` pour une liste de taille `n` $\geqslant 1$. Pour $n = 1$, le nombre d'opérations élémentaires effectuées est de 4 (1 pour `est_vide`, 1 pour `queue`, 1 pour `tête`, 1 pour la multiplication). On a donc $C_1 = 4$.

6. *Exactement* combien d'opérations élémentaires sont effectuées pour $n > 1$ pour un appel considéré seul (sans les appels engendrés) ? En déduire une relation de récurrence pour C_n pour $n > 1$.
7. Quel type de suite reconnaissez-vous ? En déduire une expression pour C_n en fonction de n.
8. En déduire, avec les notations de Landau, la complexité de cette fonction. Était-ce utile de compter exactement chaque opération élémentaire ou aurait-on pu les regrouper en un seul $\mathcal{O}(1)$?

Exercice 2 : comparaisons de complexités

1. Classez les fonctions suivantes par « ordre asymptotique » :
   $f_1(n) = n^2$, $f_2(n) = n^2 + 10^{1000} \times n$, $f_3(n) =$ si $n < 10^{1000}$ alors n^6 sinon n^2, $f_4(n) = 3^n$, $f_5(n) = n\log(n)$, $f_6(n) = n\sqrt{n}$, $f_7(n) = n^n$, $f_8(n) = n!$.
2. On note $C_1(n)$ (resp. C_2) la complexité d'une fonction $f_1(n)$ (resp. f_2). Que peut-on dire de f_1 et de f_2 si :
   - $C_1(n) = \mathcal{O}(C_2(n))$ et $C_2(n) = \mathcal{O}(C_1(n))$?
   - $C_1(n) = \mathcal{O}(C_2(n))$ et $C_1(n) = \Omega(C_2(n))$?
   - $C_1(n) = \mathcal{O}(n^k)$ et $C_2(n) = \mathcal{O}(n^k)$, avec $k \in \mathbb{N}$?
3. Soit $n \in \mathbb{N}^*$, quel est le lien entre le nombre de chiffres nécessaires pour écrire n et $\log_{10} n$? Quel rapport y a-il entre le nombre de chiffres de n et celui de $\lfloor \frac{n}{10} \rfloor$? Combien d'itérations effectue donc le code suivant ?
   ```
   i ← n
   TANT QUE i > 0
       i ← quotient de la division euclidienne de i par 10
   FIN TANT QUE
   ```
4. Avec un raisonnement similaire, donnez la complexité de la fonction OCaml suivante :
   ```
   let f n =
       let i = ref n in
       while !i > 0 do
           print_int (!i mod 2) ;
           i := !i / 2
       done
   ```

On veut savoir s'il y a des doublons dans une structure de données non triée contenant des entiers.

5. Justifier brièvement que la complexité d'une telle recherche serait quadratique en le nombre d'éléments si on ne trie pas la structure.
6. Justifier brièvement que la complexité d'une telle recherche serait linéaire en le nombre d'éléments si elle était triée.
7. Il existe des algorithmes de tris de complexité quasi-linéaire. Pour détecter les doublons, est-il rentable de commencer par trier la structure ?
8. Écrire en C la recherche de doublons dans un tableau trié, puis calculez proprement sa complexité dans le pire et dans le meilleur des cas.
9. Écrire en C la recherche de doublons dans un tableau non trié, puis calculez sa complexité dans le pire et dans le meilleur des cas.
10. Écrire en OCaml la recherche de doublons dans une *liste* triée, puis calculez sa complexité dans le pire et dans le meilleur des cas.
11. Écrire en OCaml la recherche de doublons dans une liste non triée, puis calculez sa complexité dans le pire et dans le meilleur des cas.
12. L'implémentation de la structure (avec un tableau ou avec une liste chaînée) impacte-t-elle ici l'efficacité de la recherche de doublons ?
13. Pour les algorithmes impératifs, donnez les variants et invariants permettant de justifier la terminaison et la correction de vos boucles. Pour les algorithmes récursifs, donner la propriété qu'il faudrait montrer par récurrence.
14. (Facultatif) Démontrez vos réponses à la question précédente.

Exercice 3 : fonctions impératives

1. Calculez la complexité de chacune des fonctions suivantes :

```
int f1(int n) {
    int res = 0;
    for (int i = n; i >= 1; i -= 1) {
        for (int j = 1; j <= n; j += 1) {
            res += 1;
        }
    }
    return res;
}

int f2(int n) {
    int res = 0;
    for (int i = 0; i < n; i += 1) {
        for (int j = 1; j < i; j += 1) {
            res += 1;
        }
    }
    return res ;
}

int f3(int n) {
    int res = 0;
    for (int i = n; i > 1; i = i/2) {
        res += 1;
    }
    return res;
}
```

```
int f4(int n) {
    int res = 0;
    for (int i = 1; i < n; i *= 4) {
        res += 1;
    }
    return res;
}

int f5(int n) {
    int res = 0;
    for (int i = 1; i < n; i *= 2) {
        for (int j = i; j < n; j += 2) {
            res += 1;
        }
    }
    return res ;
}

int f6(int n) {
    int res = 0;
    for (int i = 1; i <= 1000000; i += 1) {
        for (int j = i-1; j > -2; j -= 3) {
            res += 1;
        }
    }
    return res;
}
```

2. Que fait la fonction `f7` ? Calculez sa complexité en nombre de tests (= passages dans le `if`). Peut-on l'améliorer ?

```
int f7(int n) { // n >= 0 est une somme d'argent
    int res = 0 ;
    for (int i = 0; i <= n; i += 1) {
        for (int j = 0; j <= n/2; j += 1) {
            for (int k = 0; k <= n/5; k += 1) {
                for (int l = 0; l <= n/10; l += 1) {
                    for (int m = 0; m <= n/20; m += 1) {
                        if (i + 2*j + 5*k + 10*l + 20*m == n) {
                            res += 1 ;
    } } } } } }
    return res ;
}
```

Exercice 4 : fonctions récursives

Calculez la complexité de chacune des fonctions récursives suivantes :

```
let rec f1 n = match n with
    | 0 -> 1
    | _ -> 1 + f1 (n/2)

let rec f2 n = match n with
    | 0 | 1 | 2 -> n
    | _ -> 1 + f2 (n-2)

let rec f3 n = match n with
    | 0 | 1 -> n
    | _ -> f3 (n-1) + f3 (n-2)

let rec f4 n = match n with
    | _ when n < 10 -> n
    | _ when n mod 2 = 0 -> f4 (n/5)
    | _ -> f4 (n-2)

let rec f5 x n = match n with
    | 0 -> 1
    | _ -> if n mod 2 = 0 then
                f5 (x*x) (n/2)
           else
               x * f5 (x*x) (n/2)
```

```
let rec f6 n m = match n, m with
    | 0, 0 -> 0
    | 0, _ -> m + f6 n (m-1)
    | _, 0 -> n + f6 (n-1) m
    | _ -> f6 (n-1) (m-1)

let rec f7 n m = match n, m with
    | _, 1 -> false
    | 1, _ -> true
    | _ when n > m -> f7 (n-1) m
    | _ -> f7 n (m-1)

let rec f8 n m = match n with
    | _ when n = m -> 0
    | _ when n < m -> 1 + f8 (n+1) m
    | _ -> 1 + f8 (n-1) m

let rec f9 l =
    if List.length l = 0 then
        0
    else
        List.hd l + f9 (List.tl l)
```

Exercice 5 : quelques algorithmes classiques

Pour chaque algorithme que vous devez maîtriser à ce stade de l'année, écrivez en deux versions (une impérative et une récursive). Calculez soigneusement la complexité de l'intégralité de vos algorithmes. En voici quelques uns :

- Factorielle.
- Calculer le nombre de chiffres d'un entier naturel.
- Trouver le nombre d'occurrences d'un élément dans un tableau / une liste.
- Trouver le maximum / minimum dans un tableau / une liste.
- Exponentiation rapide.