TD n°3

Calcul de complexité et invariant

1 Analyse de complexité

Exercice 1 Que font les programmes suivants ? Quelle est la complexité de ces programmes dans le pire et dans le meilleur des cas ?

```
inconnue1 (T : tableau d'entiers)
début
    pour i de |T| à 2 par pas de −1 faire
        pour j de 1 à i − 1 faire
            si (T[j] > T[j + 1]) alors
                échanger T[j] et T[j + 1]
fin
```

```
inconnue2 (n : entier)
début
    pour i de 1 à n faire
        pour j de 1 à i faire
            pour k de 1 à j faire
                instruction en
                O(1)
fin
```

```
inconnue3 (a : entier)
début
    p ← 0;
    tant que (a mod 2 = 0) faire
        a ← a div 2;
        p ← p + 1;
    si (a = 1) alors
        retourner p;
    sinon
        retourner −1;
fin
```

```
inconnue4 (a, b : entiers)
début
    p ← 1;
    r ← 0;
    tant que (a > 0) faire
        r ← r + (a mod 10) × p;
        a ← a div 10;
        p ← p × b;
    retourner r;
fin
```

Exercice 2

1. Rappeler le principe du tri fusion.
2. Évaluer la complexité de cet algorithme sur une entrée de taille $n = 2^k$.

Lorsque l'on analyse la complexité d'un algorithme récursif sur un arbre binaire, on dispose dans le meilleur des cas d'une relation de récurrence de la forme $T(n) = 2T(n/2)+\Theta(f(n))$. Nous allons résoudre cette équation de récurrence dans deux cas classiques. Dans ce qui suit, a et b désignent des constantes positives.

2. Calculer la complexité d'un algorithme dont la relation de récurrence est :

$$T(n) = \begin{cases} a \text{ si } n = 1 \\ 2T(n/2) + b \text{ si } n > 1 \end{cases}$$

3. Calculer la complexité d'un algorithme dont la relation de récurrence est :

$$T(n) = \begin{cases} a \text{ si } n = 1 \\ 2T(n/2) + bn \text{ si } n > 1 \end{cases}$$

Remarque : pour mémoire, rappelons enfin la forme générale de la solution des équations de récurrence

$$T(n) = \alpha T(\frac{n}{\beta}) + f(n)$$

selon la fonction (croissante) $f(n)$. Dans ce qui suit, $\log_\beta \alpha$ dénote le logarithme de α en base β.

1. Si $f(n) = O(n^{\log_\beta \alpha - \epsilon})$ pour un certain $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_\beta \alpha})$.
2. Si $f(n) = \Theta(n^{\log_\beta \alpha})$, alors $T(n) = \Theta(n^{\log_\beta \alpha} \log n)$.
3. Si $f(n) = \Omega(n^{\log_\beta \alpha + \epsilon})$ pour un certain $\epsilon > 0$, alors $T(n) = \Theta(f(n))$.

On trouvera plus de détails à ce sujet (dont la démonstration de ce résultat) dans le livre de Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*.

2 Complexité en moyenne

Exercice 3 Soit n un entier strictement positif et soit T un tableau de n nombres distincts. On appelle *maximum provisoire* de T tout indice i dans $[1, n]$ tel que, pour tout j dans $[1, i-1]$, $T[i]$ est plus grand que $T[j]$.

On veut calculer le nombre moyen de maxima provisoires, sur tous les tableaux T qui sont des permutations de $[1, n]$ (*i.e.*, des bijections de $[1, n]$ dans $[1, n]$). C'est le nombre moyen d'a□ectations à faire dans l'algorithme classique de recherche du maximum.

On appelle $P_{n,k}$ le nombre de permutations de $[1, n]$ qui ont k maxima provisoires. Montrer que l'on a les relations :
– $P_{1,1} = 1$, et $P_{1,k} = 0$ pour tout $k \neq 1$;
– $P_{n,k} = P_{n-1,k-1} + (n-1)P_{n-1,k}$.

Soit S_n le nombre moyen de maxima provisoires d'une permutation de $[1, n]$. Montrer que :

$$S_n = H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

Exprimer S_n en Θ d'une fonction en n.

3 Invariant

Les invariants sont essentiels pour justifier la correction d'une boucle. Il s'agit d'une propriété (en général une relation entre les valeurs caractéristiques de la boucle comme son compteur, etc.) maintenue au cours du calcul, c'est-à-dire vraie au début et à la fin de chaque itération. Pour démontrer que P est un invariant, on doit vérifier que :

– P est vrai avant la première itération (pour les valeurs des variables juste avant l'entrée dans la boucle) ;
– si la condition C de boucle est satisfaite, et l'invariant est vrai pour les valeurs des variables avant l'exécution du corps de la boucle, alors il est aussi vrai pour les nouvelles valeurs des variables après une itération du corps de la boucle.

Ainsi, par récurrence, on montre que si P est bien un invariant de la boucle, lorsque le programme sort de la boucle, P est vérifié. Dans une boucle `while(C){corps de la boucle}`, C est la condition de boucle donc on aura par ailleurs $\neg C$ lorsque le programme sort de la boucle. On choisit donc la propriété P de sorte que $P \wedge \neg C$ implique la propriété de la boucle que l'on cherche à montrer. C'est ainsi que l'on justifie la correction de la boucle.

Prenons maintenant pour exemple la boucle suivante, pour laquelle on choisit l'invariant `fact` = i!, ce qui justifiera que ce programme retourne n!.

```
factorielle(n : entier)
début
    i ← 0;
    fact ← 1;
    // fact = i! verifié
    tant que (i ≠ n) faire
        i ← i + 1;
        fact ← fact × i;
        // fact = i! verifié
    // (fact = i!) ∧ (i = n) ⇒ fact = n!
    retourner fact
fin
```

Exercice 4 Que font les programmes suivants ? Justifiez votre réponse en montrant un invariant pour chaque boucle.

```
max(T : tableau d'entiers)
début
    m ← T[0];
    pour (i de 0 à |T| − 1) faire
        si (T[i] > m) alors
            m ← T[i];
    retourner m
fin
```

```
recherche(T : tableau d'entiers, e : entier)
début
    i ← 0;
    tant que (i < |T| ∧ T[i] ≠ e) faire
        i ← i + 1;
    si (i = |T|) alors
        retourner −1
    sinon
        retourner i
fin
```

Exercice 5 [Algorithme d'Euclide]

L'algorithme d'Euclide pour calculer le pgcd deux entiers a et b est basé sur la propriété suivante : si a et b sont deux entiers positifs, et si on note r_1 le reste de la division euclidienne de a par b, alors pgcd(a, b) = pgcd(b, r_1). L'idée de l'algorithme est de recommencer la même opération sur le couple (b, r_1), et ainsi de suite, jusqu'à trouver un reste r_{n+1} nul. On a alors

pgcd(a, b) = pgcd(b, r_1) = pgcd(r_1, r_2) = $\cdots$ = pgcd(r_n, 0) = r_n.

On obtient donc l'algorithme suivant.

```
pgcd(a : entier, b : entier)
début
    r ← 1;
    x ← a;
    y ← b;
    tant que (r ≠ 0) faire
        r ← x mod y;
        x ← y;
        y ← r;
    retourner x;
fin
```

Question 1. Pourquoi ce calcul termine-t-il ?

Question 2. Exhibez un invariant pour la boucle `tant que` permettant de justifier la correction du programme.

4 AVL : opérations et rotations

Exercice 6

Question 1. (Rotation Simple) Donner l'AVL obtenu en insérant successivement les valeurs : 1,2,3,4,21 .

Question 2. (Rotation Double) Insérer maintenant : 22,23,24, 11,12,13 14

Question 3. Donner l'AVL obtenu en supprimant 4 de l'AVL calculé précédemment.