TP 5 : Complexité

Semaine du 9 Octobre

Exercice 1. Domination

Pour chacune des fonctions suivantes, proposer une fonction simple qui la domine. On veillera à prendre une fonction la plus petite asymptotiquement possible.

1. $f_1(n) = 3n^2$
2. $f_2(n) = 2n + n^2$
3. $f_3(n) = \frac{n(n-1)}{2}$
4. $f_4(n) = n^{100} + \left(\frac{101}{100}\right)^n$
5. $f_5(n) = n \log n + n^2$
6. $f_6(n) = n^{1.01} + n \log n$
7. $f_7(n) = n \log n + \frac{n^2}{\log n}$

Exercice 2. Complexité

Pour chacune des fonctions suivantes, proposer une complexité asymptotique :

1.
```
let f n =
    for i = 0 to n-1 do
        print_int i ; print_newline ()
    done
```

2.
```
let rec f n = match n with
    | 0 -> 1
    | _ -> n * f (n - 1)
```

3.
```
let f n =
    for i = 0 to n-1 do
        print_int i ; print_newline ()
    done ;
    for i = 0 to n-1 do
        for j = 0 to n-1 do
        print_int (i*j) ; print_newline()
    done
```

4.
```
let f n =
    let rec aux n k = match k with
    | 0 -> 0
    | _ -> n * k + aux n (k-1)
    in let a = ref 0 in
    for i = 0 to n do
        a:= !a + aux n i
    done ; !a
```

5.
```
let rec f n =
    let res = ref 0 in
    for i = 0 to n-1 do
        for j = i to n-1 do
            if j mod i = 0 then
                res:= !res + 1
        done
    done ; !res
```

6.
```
let rec f n = match n with
    | 0 -> 1
    | _ -> f (n-1) + f (n-1)
```

7.
```
let f n =
    let rec aux1 k = match k with
    | 0 -> 0
    | _ -> k + aux1 (k-1)
    and aux2 k = match k with
    | 0 -> 0
    | _ -> aux1 k + aux2 (k-1)
    in aux2 n
```

8.
```
let rec f n = match n with
    | 0 -> 1
    | _ when n mod 2 = 0 -> let temp = f (n/2) in temp * temp
    | _ -> let temp = f (n/2) in temp * temp * 2
```

9.
```
let rec rajouter liste x = match liste with
| [] -> [[x]]
| p::q -> (x::p)::rajouter q x
```

10.
```
let f n =
    let rec range n = match n with
    | 0 -> [0]
    | _ -> n::range (n-1)
    and rajouter liste x =
    match liste with
    | [] -> [[x]]
    | p::q -> (x::p)::rajouter q x
    and parties liste =
    match liste with
    | [] -> []
    | p::q -> let sans_p = parties q in
        let avec_p = rajouter sans_p x in
        sans_p@avec_p
    in parties (range n)
```

Exercice 3. Concaténation et complexité

1. Sans utiliser l'opérateur `@`, proposer une implémentation d'une fonction `concat` qui concatène deux listes.
2. On s'intéresse à la fonction suivante :

```
let rec inverse l = match l with
| [] -> []
| p::q -> concat (inverse q) [p]
```

   Quelle est sa complexité en terme d'opérations élémentaires de cette fonction ?
3. Proposer une implémentation de la fonction `inverse` qui n'utilise pas `concat` dont la complexité est en $O(n)$ où n est la taille de la liste en argument.

Exercice 4. Complexité du calcul de Fibonacci

On se donne la suite de Fibonacci définie comme suis :

$$u_0 = 0$$
$$u_1 = 1$$
$$\forall n \in \mathbb{N}\ u_{n+2} = u_{n+1} + u_n$$

On propose l'implémentation suivante dont on cherche à calculer la complexité :

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

1. Combien d'appels à fibonacci ont lieu lors du calcul de `fibonacci 2` ? De `fibonacci 4` ? Représenter les appels récursif sous la forme d'un arbre.
2. Montrer que le nombre d'appels à la fonction `fibonacci` dans le calcul de `fibonacci n` noté $C(n)$ vérifie la formule de récurrence suivante :

$$\forall n \geqslant 2, C(n) = 1 + C(n-1) + C(n-2)$$

3. Montrer par récurrence que pour tout $n \geqslant 0$, $C(n) = 2u_{n+1} - 1$.
4. On admet que, pour tout $n \geqslant 0$, on a :

$$u_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

   En déduire la complexité asymptotique de `fibonacci`.
5. Proposer une implémentation de `fibonacci` en $O(n)$.

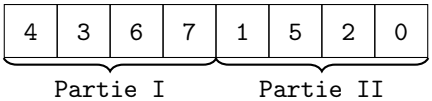
Exercice 5. Trouver une porte face à un mur dans l'obscurité

On considère un mur de longueure infinie, on peut se déplacer le long du mur dans un sens ou dans un autre, et l'objectif est de trouver une porte située d'un côté ou de l'autre à une distance l inconnue. À cause de l'obscurité, on ne peut trouver la porte que lorsqu'on est directement face à elle.

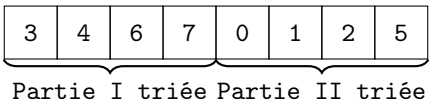
— Proposer une méthode pour trouver la porte.
— Proposer une méthode pour trouver la porte telle que la distance parcourue soit en $O(l)$.

Exercice 6. Tri fusion et complexité

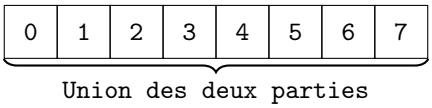
Le tri fusion est un tri de type diviser-pour-règner rapide et relativement facile à implémenter. Il consiste à diviser le tableau en deux, de trier chaque partie récursivement, puis d'unir les deux parties du tableau en remarquant qu'il est facile de créer un tableau ordonné à partir de deux parties ordonnées.



Après les appels récursifs, on obtient :



Après l'union des deux sous-tableaux triés, on obtient :



Pour l'union, l'idée est de copier les deux sous-tableau autre part et de parcourir les deux sous-tableaux déjà triés en parrallèle et de choisir à chaque fois l'élément le plus petit pour construire l'union des deux.

1. Proposer une fonction `unir` de signature `'a array -> int -> int -> int ->unit` telle que `unir t i j k`, sous réserve que les sous-tableaux dont les éléments sont à l'indices i jusqu'à l'indice j et de $j+1$ jusqu'à l'indice k soient triés, modifie le tableau t de sorte à ce que les éléments de i à k soient triés par ordre croissant.

   On pourra créer un autre tableau pour stocker les données du premier sous-tableau.
2. En déduire une implémentation du tri fusion.
3. En notant $C(n)$ la complexité en affectation du tri fusion, montrer que $C(n)$ vérifie la propriété :

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

4. En déduire que $C(n) = O(n \log n)$.

   On pourra supposer que n est une puissance de deux.

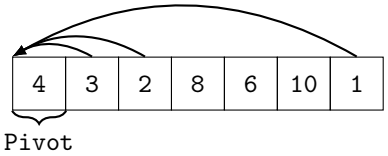
Exercice 7. Pré-calcul pour le nombre de doublons dans un tableau

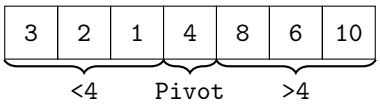
1. Proposer une fonction en $O(n^2)$ qui compte le nombre de doublons dans un tableau, c'est-à-dire le nombre d'éléments qui sont présents plus d'une fois dans un tableau.
2. En utilisant le tri fusion, proposer une fonction en $O(n \log n)$ qui compte le nombre de doublons.

Exercice 8. Tri rapide et Complexité

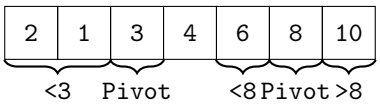
Le tri rapide est l'un des tri les plus utilisés, car il est effectivement très rapide *en moyenne*.

À chaque étape, on choisit un pivot, par exemple le premier élément, puis on modifie le tableau en le parcourant de sorte à avoir tout les éléments plus petits que le pivot à gauche de celui-ci, et les éléments plus grand à droite du pivot.





On applique ensuite récursivement notre algorithme sur les parties gauches et droite du tableau.



Après une dernière étape, on obtient le tableau trié.

1. Montrer que la complexité dans le meilleur des cas du tri rapide est en $O(n \log n)$. On pourra supposer que pour un tableau de taille n, le pivot choisi discrimine le tableau en un tableau de taille $\lceil \frac{n-1}{2} \rceil$ et $\lfloor \frac{n-1}{2} \rfloor$.
2. Quelle est la complexité dans le pire des cas du tri rapide ? Proposer une famille de tableau qui atteigne cette complexité.
3. Proposer une fonction récursive `pivot` de signature `'a array -> int -> int -> unit` qui prend en entrée un tableau t, et deux entiers i et j, qui utilise `t.(i)` et qui modifie le tableau de sorte à ce que les éléments d'indices i à j soient déplacés pour avoir tous les éléments inférieurs au pivot, puis le pivot, puis tous les éléments supérieurs au pivot.

   Cette fonction s'appelle ensuite récursivement sur les deux sous-tableaux non-vides à gauche et à droite du pivot.
4. En déduire une implémentation du tri rapide.

   On cherche à montrer que la complexité du tri rapide est en $O(n \log n)$ dans le cas moyen. On suppose que le tableau est une permutation aléatoire.
5. (a) Montrer que la complexité en nombre de comparaison en fonction de la taille du tableau n du tri rapide moyen $C(n)$ vérifie la propriété suivante :

$$C(n) = \frac{1}{n}\left(\sum_{k=0}^{n-1} C(k) + C(n-1-k)\right) + n - 1$$

   (b) Montrer que, pour tout $n > 0$:

$$C(n) = \frac{2}{n}\left(\sum_{k=0}^{n-1} C(k)\right) + n - 1$$

   (c) Montrer que, pour tout $n > 0$:

$$nC(n) - (n-1)C(n-1) = 2C(n-1) + O(n)$$

   (d) Montrer que, pour tout $n > 0$:

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = O(n)$$

   (e) Montrer qu'il existe $K > 0$ tel que pour pout $n > 0$:

$$C(n) \leqslant Kn \sum_{k=1}^{n} \frac{1}{k}$$

   (f) Conclure.

Il existe des solutions pour améliorer le pire des cas du tri rapide, ou tempérer la probabilité du pire des cas. Nous verrons cela plus tard dans l'année.

Exercice 9. Bogo-tri

Le bogo-tri, ou tri stupide, est un tri particulièrement idiot qui consiste à trier un tableau en répétant deux étapes :

— On vérifie si le tableau est trié, et dans ce cas, on s'arrête ;
— Sinon, on mélange le tableau au hasard, et on recommence.

1. Quelle est la complexité en terme de comparaison et d'affectation dans le meilleur des cas ? Quelle est la probabilité que ce cas ait lieu ?
2. Quelle est la complexité dans le pire des cas ? Quelle est la probabilité que ce cas ait lieu ?
3. Proposer une fonction `verifier` de signature `'a array -> bool` qui vérifie si un tableau est trié. On interrompera le calcul dès qu'on pourra conclure.
4. Quelle est la complexité dans le meilleur des cas de votre fonction `verifier` en terme de comparaison ? Dans le pire des cas ?
5. OCaml permet de générer un nombre au hasard grâce à l'appel `Random.int n` qui renvoie un entier au hasard entre 0 et $n-1$ inclus. On aura besoin de faire un appel à `Random.self_init ()` dans un premier temps pour s'assurer que la fonction aléatoire est proprement définie.

   Proposer une implémentation du Bogo-tri.