Algorithmique

TD3 : Complexité et récursivité

1 Calculs de complexité

Évaluer la complexité des algorithmes suivants. On suppose que le coût de `OpérationElémentaire()` est $O(1)$, tandis que le coût de `Puissance(`x`,`n`)` est $O(\log n)$.

Algorithme 1 : Disjonction asymétrique

```
Entrées : n, m: entiers
pour i de 1 à n faire
    si i est pair alors
        OpérationElementaire()
    sinon
        pour j de 1 à m faire
            OpérationElementaire()
```

Algorithme 2 : Somme de puissances

```
Entrées : n: entier
i ← 1
tant que i ≤ n faire
    pour j de 1 à i faire
        OpérationElementaire()
    i ← i * 2
```

Algorithme 3 : Croissance très rapide

```
Entrées : n: entier
i ← 2
tant que i ≤ n faire
    OpérationElementaire()
    i ← i^2
```

Algorithme 4 : Condition coûteuse

```
Entrées : n:entier
i ← 0
tant que Puissance(2,i) < n faire
    i ← i + 1
retourner i
```

Algorithme 5 : Croissance non triviale

```
Entrées : n: entier
i ← 0, j ← 1
tant que i ≤ n faire
    OpérationElementaire()
    i ← i + j
    j ← j + 1
```

Algorithme 6 : Calcul corsé

```
Entrées : n: entier
i ← n, j ← 0
tant que i ≥ 1 faire
    pour k de 1 à i faire
        pour ℓ de 1 à j faire
            OpérationElementaire()
    i ← i/2
    j ← j + 1
```

2 Recherche dichotomique

Etant donné un tableau `tab` trié dans l'ordre croissant, et une valeur `x`, la recherche dichotomique de `x` dans `tab` fonctionne comme suit. L'élément recherché `x` est comparé avec l'élément médian `y`. S'ils sont égaux, on renvoie `Vrai`. Si $x < y$ on continue la recherche dichotomique sur la moitié gauche de `tab`, et si $x > y$ on continue la recherche dichotomique sur la moitié droite de `tab`, en répétant ce même procédé. L'Algorithme `EstPrésent` décrit ci-après en donne une implémentation possible.

1. Donner un invariant de boucle vérifié par `EstPrésent` qui permet de garantir que l'algorithme est correct.

   Indice : Cet invariant concerne les valeurs de `tab[début]`, `tab[fin]`, et `x`.

2. Donner un invariant de boucle vérifié par `EstPrésent` qui permet de garantir que l'algorithme est de complexité $O(\log n)$, puis justifier le calcul de cette complexité.

   Indice : Cet invariant concerne l'évolution des valeurs de `début` et `fin`.

Algorithme 7 : EstPrésent

Entrées : x: valeur, `tab`: tableau trié croissant de taille n
si $x <$ *tab[0]* **ou** $x >$ *tab[n-1]* **alors**
  retourner *Faux*
début $\leftarrow 0$, fin $\leftarrow n-1$
tant que *fin* $\geq$ *début* **faire**
  milieu $\leftarrow \lfloor$(début+fin)/2$\rfloor$
  si *tab[milieu]* $= x$ **alors**
    retourner *Vrai*
  si *tab[milieu]* $> x$ **alors**
    fin $\leftarrow$ milieu $- 1$
  sinon
    début $\leftarrow$ milieu $+ 1$
retourner *Faux*

3. Ecrire un algorithme `PlusProche(tab,x)` qui renvoie la valeur contenue dans `tab` la plus proche de `x` (i.e. la valeur $y \in$ `tab` telle que $|y-x|$ est minimum). Celui-ci devra reposer sur une recherche dichotomique, et donc être de complexité $O(\log n)$.

4. Ecrire une fonction `ISqrt(n)` qui prend en entrée un entier n, et renvoie $\lfloor\sqrt{n}\rfloor$, en utilisant une recherche dichotomique (et uniquement des opérations arithmétiques : addition, soustraction, multiplication, division entière).

3 Fibonacci

Les nombres de Fibonacci sont définis par $F_0 = F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$ pour tout $n \geq 2$. On considère l'algorithme suivant.

Algorithme 8 : Fibo

Entrées : $n \geq 0$: entier
si $n \leq 1$ **alors**
  retourner *1*
sinon
  retourner *Fibo(n-1)+Fibo(n-2)*

1. Énumérer les appels récursifs de `Fibo(5)` (sous la forme d'un arbre). Que remarque-t-on ?

2. La complexité de l'algorithme `Fibo` est exponentielle, à cause de redondances dans les appels récursifs. Pour pallier cela, on peut implémenter un algorithme récursif `Fibo2` qui prend comme paramètre un entier n, et qui retourne la paire (F_n, F_{n+1}). Écrire un tel algorithme.

3. Quelle est la complexité de `Fibo2` ?