Correction de TD
Récursivité

Ce document contient quelques éléments de correction pour les TD. Il n'est pas voué à être exhaustif et ne se substitue **pas** à la correction vue en TD.

Questions sur le travail en autonomie : quelles sont les complexités temporelles de
— l'algorithme d'Euclide ? On doit compter le nombre d'étapes dans la boucle, chaque étape étant en temps constant. On voit facilement qu'on est en $O(n)$ puisqu'entre chaque étape de la boucle, les variables diminuent strictement (et ce sont des entiers). Plus fin, on peut même dire qu'on est en $O(\log(n))$: en effet, on remarque que chaque variable est divisée par au moins 2 toutes les deux étapes.
— l'algorithme calculant le n-ième terme de la suite de Syracuse ? On ne risque pas de pouvoir exprimer une complexité temporelle : il n'a jamais été prouvé que cet algorithme termine, c'est à dire que pour toute valeur initiale, la suite de Syracuse atteint la valeur 1 à partir d'un certain rang.

1 Fibonacci

On veut calculer le n-ième terme f_n de la suite de Fibonacci définie par :

$$f_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } 1 \\ f_{n-1} + f_{n-2} & \text{sinon} \end{cases}$$

Notez qu'on prend la convention $f_0 = f_1 = 1$ mais une autre convention usuelle consiste à prendre $f_0 = 0$, ce qui ne fait que décaler les termes de la suite.

Vous avez vu en cours qu'un algorithme naïf récursif est exponentiel. On propose d'abord un algorithme itératif que l'on adapte ensuite pour trouver une version récursive. Si on connaît deux termes consécutifs de la suite (f_i, f_{i+1}), alors on peut en déduire (f_{i+1}, f_{i+2}). Il suffit donc de partir de (f_0, f_1) et d'avancer jusqu'à (f_{n-1}, f_n). Cet algorithme est en $O(n)$.

Algorithm 1: FiboIteratif

Input: n un entier naturel
$f_{k-1} \leftarrow 1$
$f_k \leftarrow 1$
for k *de* 2 *à* n **do**
  $tmp \leftarrow f_k$
  $f_k \leftarrow f_k + f_{k-1}$
  $f_{k-1} \leftarrow tmp$
end
return f_k

On propose maintenant une version récursive. Il faut avoir en tête que si l'on fait l'appel `FiboRec`(k, x, y), alors $f_n = x \cdot f_k + y \cdot f_{k-1}$. L'algorithme 3 lance le premier appel de `FiboRec` avec la bonne initialisation puisque $f_n = 1 \cdot f_n + 0 \cdot f_{n-1}$ Cet algorithme est récursif terminal et est en $O(n)$.

Algorithm 2: FiboRec

Input: n, x, y des entiers naturels

```
1 if n ≤ 1 then
2 |  return x + y
3 end
4 else
5 |  return FiboRec(n − 1, x + y, x)
6 end
```

Algorithm 3: Fibo

Input: n un entier naturel

```
1 return FiboRec(n, 1, 0)
```

2 Somme des chiffres

On se donne un nombre entier naturel et on veut calculer la somme des chiffres qui apparaissent dans son écriture décimale. Par exemple, si on se donne 1402, on veut que notre algorithme renvoie 7. Pour cela, on applique l'algorithme récursif suivant. On récupère le dernier chiffre en prenant le modulo 10, puis on fait un appel récursif sur x privé de son dernier chiffre, i.e. $\left\lfloor \frac{n}{10} \right\rfloor$. Cet algorithme est en $O(\log n)$ puisque x contient $\lfloor \log_{10}(n) \rfloor + 1$ chiffres. Il est récursif mais pas récursif terminal puisqu'on doit sommer en remontant les appels.

Algorithm 4: SommeChiffreRec

Input: n un entier naturel

```
1 if n = 0 then
2 |  return 0
3 end
4 else
5 |  return (n mod 10)+ SommeChiffreRec(⌊n/10⌋)
6 end
```

Afin de donner une version récursive terminale, on peut faire passer la partie $(x \mod 10)$ par paramètre. On lance le premier appel avec l'algorithme 6.

Algorithm 5: SommeChiffreRecTerminale

Input: n un entier naturel, *sommePartielle* somme des chiffres déjà vus

```
1 if n = 0 then
2 |  return sommePartielle
3 end
4 else
5 |  return SommeChiffreRecTerminale(⌊n/10⌋, sommePartielle + (n mod 10))
6 end
```

Algorithm 6: SommeRecTerminaleInit

Input: n un entier naturel

return SommeChiffreRecTerminale$(n, 0)$

3 Inversion

On se donne un nombre entier naturel et on veut inverser son écriture décimale. Par exemple, si on se donne 1907, notre algorithme doit renvoyer 7091.

On propose tout d'abord une version récursive non terminale. Cette dernière récupère le chiffre de droite (comme pour l'exercice précédent) et le place directement en bonne position en le multipliant par la bonne puissance de 10 : on a vu dans l'exercice précédent que le logarithme en base 10 donne le nombre de chiffre en décimal.

Algorithm 7: InvertRec

Input: n un entier naturel

if $n = 0$ **then**

  return 0

end

else

  return Invert$\left(\left\lfloor \frac{n}{10} \right\rfloor\right) + (n \mod 10) \cdot 10^{\lfloor \log_{10}(x) \rfloor}$

end

On propose ensuite une version récursive terminale qui nous évite de placer directement à gauche le chiffre de droite. On le place à droite de *seen*, et il sera décalé vers la gauche à chaque appel récursif puisqu'on multiplie par 10. L'initialisation est faite avec l'algorithme 9

Algorithm 8: InvertRec

Input: $notSeen, seen$ deux entiers naturels

if $notSeen = 0$ **then**

  return 0

end

else

  return InvertRec$\left(\left(\left\lfloor \frac{notSeen}{10} \right\rfloor\right), 10 \cdot seen + (notSeen \mod 10)\right)$

end

Algorithm 9: Invert

Input: n un entier naturel

return Invert$(n, 0)$

Vous pouvez faire tourner l'algorithme sur l'entrée 1907 : le Tableau 1 décrit l'évolution de *seen* et *notSeen* pour chaque appel récursif.

$notSeen$	$seen$
1907	0
190	7
19	70
1	709
0	7091

TABLE 1 – Exemple d'exécution d'InvertRecTerminale pour 1907.

4 Recherche dichotomique

On se donne un tableau T d'entiers trié et un entier x. On veut vérifier si x est dans T. Pour cela, on peut faire un parcours linéaire du tableau. Cependant, le tableau étant trié, on peut proposer une méthode beaucoup plus efficace basée sur l'idée suivante. Soit m l'élément au milieu du tableau. Si $x < m$, il suffit de chercher dans la moitié gauche du tableau ; sinon, dans la moitié droite. Avec un seul test, on divise par deux le nombre de positions à vérifier dans le tableau. En appliquant récursivement ce procédé, on obtient un algorithme en $O(\log n)$, avec n la taille du tableau.

L'algorithme 10 applique ce procédé sur le sous tableau allant des indices min à max. On initialise avec l'algorithme 11, de 0 à $n-1$, c'est à dire tout le tableau.

Algorithm 10: RechercheDichoRec

Input: T un tableau de réels, min, max deux entiers, x un nombre réel

if $T\left[\left\lfloor \frac{min+max}{2} \right\rfloor\right] = x$ **then**
  return True
else if $min = max$ **then**
  return False
else if $T\left[\left\lfloor \frac{min+max}{2} \right\rfloor\right] = x$ **then**
  return RechercheDichoRec$\left(T, min, \left\lfloor \frac{min+max}{2} \right\rfloor\right)$
else
  return RechercheDichoRec$\left(T, \left\lfloor \frac{min+max}{2} \right\rfloor + 1, max\right)$

Algorithm 11: RechercheDicho

Input: T un tableau de réels, n la taille d'un tableau, x un nombre réel

return RechercheDicho$(T, 0, n-1, x)$

5 Placement de reines

On veut afficher toutes les possibilités de placer n reines sur un échiquier ayant n^2 cases, sans qu'elles ne puissent se manger. On rappelle qu'une reine peut manger sur toute sa ligne, sa colonne et ses diagonales. On propose un algorithme sans les détails, ces derniers étant discutés par la suite.

On remarque déjà qu'il n'y qu'une reine par ligne. On va donc placer les reine successivement sur les lignes 0, puis 1, etc... L'appel récursif avec i en numéro de ligne essaie de placer une reine sur la ligne i. On teste pour toutes les colonnes. Etant donnée une colonne j, si le placement est correct (pas de reine sur la même colonne ou sur une même diagonale), alors on continue récursivement sur la ligne suivante $(i+1)$. Cet appel cherche récursivement tous placements valides avec les reines des lignes 0 à i déjà placées. Une fois que l'on remonte de cet appel récursif, on teste de placer la reine de la i-ème ligne sur la colonne suivante $j+1$, etc...

Enfin, quand $i = taille(grille)$, c'est qu'il y a une reine par ligne : plus moyen d'en ajouter. Donc on peut afficher la grille car on a placé n reines.

L'algorithme 13 initialise la grille vide, puis fait un appel à l'algorithme récursif 12 en commençant par remplir la ligne 0.

Algorithm 12: SolveurReineRec

Input: *grid* une grille de taille $n \times n$, i une ligne

```
1  if i = n then
2  |  afficher(grid)
3  end
4  else
5  |  if placementCorrect(grid, i, j) then
6  |  |  for j de 0 à taille(grid) do
7  |  |  |  placerReine(grid, i, j)
8  |  |  |  SolveurReineRec(grid, i + 1)
9  |  |  |  retirerReine(grid, i, j)
10 |  |  end
11 |  end
12 end
```

Algorithm 13: SolveurReine

Input: n un entier naturel

```
1  grid ← grille vide de taille n × n SolveurReineRec(grid, 0)
```

On définit les fonctions utilisées dans l'algorithme 12 :

placementCorrect prend une grille, une ligne i et une colonne j en arguments et renvoie un booléen qui est Vrai si et seulement si on peut rajouter une reine dans la grille en position (i, j) sans qu'aucune reine déjà placée ne puisse la manger.

placerReine prend une grille, une ligne i et une colonne j en arguments et place une reine dans la grille à la position (i, j).

retirerReine prend une grille, une ligne i et une colonne j en arguments et retire une reine dans la grille à la position (i, j).

On peut ensuite réfléchir à l'implémentation de la structure de grille et des fonctions ci-dessus. Premièrement, la grille peut contenir des booléens : une case contient Vrai si et seulement si il y a une reine dessus. Ensuite, une grille est un tableau bidimensionnel. On peut l'implémenter avec un tableau unidimensionnel T. Il faut juste pouvoir convertir les indices de la grille vers les positions dans T. Pour le placement ou le retrait d'une reine, il suffit de changer le booléen dans la case correspondante.

Pour le test de placement, on peut regarder si une case sur la même colonne ou même diagonale contient Vrai. Pour la colonne, on peut vérifier si tous les booléens de la même colonne sont à False. Plus malin, on initialise un tableau $Column$ de booléen de taille n au début. $Column[j]$ contient Vrai si et seulement si il y a une reine sur la colonne j. A chaque ajout de reine, on peut mettre ce tableau à jour en temps constant (de même pour le retrait car il y a au plus une seule reine par colonne). Ce tableau permet de vérifier si une colonne est déjà prise en temps constant. On peut faire pareil pour les $2n-1$ diagonales dans un sens et les $2n-1$ dans l'autre sens.

Question subsidiaire : montrer que si $n > 3$, il existe toujours un placement valide de n reines.