Correction de TD
Révisions sur les tableaux

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.

1 Décallage dans un tableau

On cherche à décaler vers la droite les éléments d'un tableau à partir de celui d'indice k. On peut parcourir itérativement les éléments du tableau, soit de la droite vers la gauche, soit l'inverse. La deuxième méthode, quoique plus naturelle, nécessite l'utilisation de deux variables temporaires.

Algorithm 1: Décallage d'un tableau par la droite

Input: T un tableau, n sa taille, k une position
for i *de* $n-1$ *à* $k+1$ **do**
  $T[i] \leftarrow T[i-1]$
end

Algorithm 2: Décallage d'un tableau par la gauche

Input: T un tableau, n sa taille, k une position
$tmp1 \leftarrow T[k]$
$tmp2 \leftarrow T[k+1]$
for i *de* $k+1$ *à* $n-2$ **do**
  $T[i] \leftarrow tmp1$
  $tmp1 \leftarrow tmp2$
  $tmp2 \leftarrow T[i+1]$
end
$T[n-1] \leftarrow tmp1$

La complexité des deux algorithmes est linéaire, i.e. en $O(n)$. Plus précisement, le premier algorithme demande $n-k-1$ opérations, tandis que le deuxième en demande $3(n-k-2)+1$. Ces algorithmes sont utiles si on veut insérer un élément au milieu d'un tableau trié par exemple.

2 Tableaux de présence

On se donne deux tableaux de booléens, un pour chaque devoir. $DS1[i]$ contient Vrai si l'étudiant i étant présent au premier devoir, Faux sinon. Pareil pour $DS2$.

2.1 Toujours présent

Pour chaque étudiant, il est toujours présent si et seulement si il est présent au premier **et** au deuxième DS. Pour calculer le tableau contenant cette information, il suffit de parcourir les deux tableaux $DS1$ et $DS2$ en parallèle.

Algorithm 3: ToujoursPrésent

Input: $DS1, DS2$ deux tableaux de booléens de taille n

```
1 T ← tableau de booléens de taille n
2 for i de 0 à n − 1 do
3 |  T[i] ← DS1[2] et DS2[i]
4 end
5 return T
```

2.2 Parfois présent

Pour chaque étudiant, il n'est pas toujours absent si et seulement si il est présent au premier **ou** au deuxième DS.

Algorithm 4: PasToujoursAbsent

Input: $DS1, DS2$ deux tableaux de booléens de taille n

```
1 T ← tableau de booléens de taille n
2 for i de 0 à n − 1 do
3 |  T[i] ← DS1[2] ou DS2[i]
4 end
```

3 Recherche de somme

Pour une valeur s donnée, on cherche un couple $(T[i], T[j])$ d'éléments d'un tableau tels que $T[i] + T[j] = s$ et $i \neq j$. On propose d'abord de regarder tous les couples d'éléments du tableau.

Algorithm 5: coupleDeSommeNaïf

Input: T un tableau d'entiers de taille n, s un entier

```
1  T ← tableau de booléens de taille n
2  for i de 0 à n − 1 do
3  |  for j de 0 à n − 1 do
4  |  |  if i ≠ j then
5  |  |  |  if T[i] + T[j] = s then
6  |  |  |  |  return Vrai
7  |  |  |  end
8  |  |  end
9  |  end
10 end
11 return Faux
```

Cet algorithme est en n^2 étapes : il y a n^2 couples (i, j) et chaque étape de boucle est en temps constant.

Algorithm 6: coupleDeSommeUnPeuMoinsNaïf

```
Input: T un tableau d'entiers de taille n, s un entier
for i de 0 à n − 1 do
    for j de i + 1 à n − 1 do
        if T[i] + T[j] = s then
            return Vrai
        end
    end
end
return Faux
```

On conserve un algorithme quadratique, i.e. en $O(n^2)$ mais plus exactement en $\frac{n(n-1)}{2}$ étapes cette fois. Deux façons de le voir :

- Pour chaque valeur de i, j peut prendre $n-i-1$ valeurs. Pour avoir le nombre d'étapes dans la double boucle, il faut donc sommer sur tous les i : $\sum_{0 \leq i \leq n-1} n-i-1 = \sum_{0 \leq i \leq n-1} i = \frac{n(n-1)}{2}$
- Sur les n^2 étapes de l'algorithme précédent, on a retiré les n cas où $i = j$, et parmi les cas où $i \neq j$, on ne traite que ceux où $i < j$, donc la moitié. Ainsi, il y a la moitié de $n^2 - n$, soit $\frac{n(n-1)}{2}$.

On propose une méthode plus rapide : on fait un prétraitement en triant le tableau, puis un algorithme linéaire suffit.

Algorithm 7: coupleDeSommeAvecTri

```
Input: T un tableau d'entiers de taille n, s un entier
T ← tri(T)
i ← 0
j ← n − 1
while i ≠ j do
    if T[i] + T[j] = s then
        return Vrai
    else if T[i] + T[j] < s then
        i ← i + 1
    else
        j ← j − 1
end
return Faux
```

Avec les tris fusion et rapide vu en L1, on peut obtenir une complexité en $O(n \log n)$. Ensuite, chaque étape de la boucle while est en temps constant. Enfin, à chaque étape de la boucle, i croît ou j décroît. Donc il y a au plus n étapes. Ainsi, une fois le tableau trié, le reste de l'algorithme est en $O(n)$. Donc l'algorithme est en $O(n \log n)$.

4 Indice séparateur

On cherche un indice d'un tableau tel que la somme des éléments d'indices strictement inférieurs soit égale à celle des éléments d'indices strictement supérieurs. Cet indice est appelé un indice séparateur. Un premier algorithme naïf teste indépendamment si chaque indice du tableau est un indice séparateur. SommeSousTableau(T, i, j) calcule la somme des éléments de T de la position i à la position j.

Algorithm 8: IndiceSepNaïf

```
   Input: T un tableau d'entiers de taille n
1  for i de 0 à n − 1 do
2  |  if SommeSousTableau(T, 0, i − 1) =SommeSousTableau(T, i + 1, n − 1) then
3  |  |  return i
4  end
5  return Faux
```

Algorithm 9: SommeSousTableau

```
   Input: T un tableau d'entiers de taille n, i, j deux positions du tableau
1  sum ← 0 for k de i à j do
2  |  sum ← sum + T[k]
3  end
4  return sum
```

Plus malin, on peut éviter de recalculer les sommes à chaque étape, obtenant ainsi un algorithme linéaire.

Algorithm 10: IndiceSepLineaire

```
   Input: T un tableau d'entiers de taille n
1  somme ← 0
2  for i de 0 à n − 1 do
3  |  somme ← T[i]
4  end
5  sommeGauche ← 0
6  sommeDroite ← somme − T[0]
7  if sommeDroite = 0 then
8  |  return 0
9  end
10 for i de 0 à n − 2 do
11 |  sommeGauche ← sommeGauche + T[i]
12 |  sommeDroite ← sommeDroite − T[i + 1]
13 |  if sommeGauche = sommeDroite then
14 |  |  return i
15 |  end
16 end
17 return Faux
```