



SORBONNE UNIVERSITÉ

3I003 : ALGORITHMIQUE

Projet :  
**Confitures**

*Ahmed Boukerram*  
*Angelo Ortiz*

Licence d'Informatique  
Année 2018/2019

# Table des matières

1	Introduction	2
I	Partie théorique	3
2	Algorithme I : Recherche exhaustive	3
3	Algorithme II : Programmation dynamique	6
4	Algorithme III : Cas particulier et algorithme glouton	11
II	Mise en œuvre	12
5	Implémentation	12
Annexe A	Suites croissantes	16
Annexe B	Complexité de RechercheExhaustive dans le cas général	18

# 1 Introduction

Sur un forum de cuisine, on pourrait trouver une question portant sur le nombre minimal de pots nécessaires pour la préservation d'une certaine quantité de confiture préparée. Il est important de remplir au complet chaque bocal afin de prolonger au maximum le temps de conservation du produit. Mais la capacité des bocaux n'est pas la même pour tous. C'est pourquoi on veut proposer des algorithmes résolvant cette question : on se fixe pour objectif d'utiliser le moins possible de bocaux.

Pour définir ce problème, on dispose des données suivantes. Il y a  $S$  décigrammes de confiture que l'on doit verser dans des bocaux vides. On compte aussi plusieurs bocaux de diverses capacités que l'on range en  $k$  classes : chaque classe de bocal correspond à une capacité distincte. On appellera *système de capacités* l'ensemble des capacités dont on dispose. On note ces  $k$  capacités par un tableau  $V$  de taille  $k$  numéroté par ordre croissant :

- $V[1] < V[2] < \dots < V[k]$  ; et
- chaque capacité  $V[i]$  est exprimée par la quantité en décigrammes que l'on peut mettre dans un bocal.

On se fixe aussi certaines contraintes qui garantissent l'existence d'une solution au problème :

- la quantité  $S$  est un nombre entier de décigrammes ;
- la capacité minimal d'un bocal est 1 décigramme, i.e.  $V[1] = 1$  ; et
- on dispose d'une très grande quantité (supposée ainsi illimitée) de bocaux de chacune des capacités.

En effet, en tenant compte des trois hypothèses, on remarque que l'on peut toujours verser la totalité de confiture dans des pots de 1 décigramme.

Notre objectif est de remplir le moins possible de bocaux et qu'ils soient tous remplis exactement à sa capacité maximale. Ainsi, étant donnés :

- un système de  $k$  capacités  $V[i] \in \mathbb{N}$ ,  $i \in \{1, \dots, k\}$  avec  $V[1] = 1$ ,
- et une quantité totale  $S \in \mathbb{N}$  de confiture,

le but est de déterminer le nombre minimum de bocaux tels que la somme de leurs capacités est égale à  $S$ . On cherche donc à retourner un couple  $(n, A)$ , où  $n$  est le nombre de bocaux utilisés et  $A$  est un tableau de taille  $k$  tel que  $A[i]$  représente le nombre de bocaux de capacité  $V[i]$  à remplir au complet. On a ainsi que  $n = \sum_{i=1}^k A[i]$ .

Les objectifs de ce projet sont l'analyse théorique et la mise en œuvre de trois algorithmes résolvant le problème décrit ci-dessus. Dans un premier temps, on formalise le problème et propose plusieurs algorithmes dont on fera l'analyse de complexité. Dans un deuxième temps, on implémente ces algorithmes au moyen du langage de programmation Python afin de vérifier expérimentalement les complexités trouvées précédemment.

# Première partie

## Partie théorique

### 2 Algorithme I : Recherche exhaustive

#### Question 1

Soit la propriété suivante  $P_1(s)$  : `RechercheExhaustive`( $k, V, s$ ) se termine et renvoie le nombre minimal de bocaux à remplir pour une quantité  $s$  de confiture avec un système de  $k$  capacités décrit dans les cases du tableau  $V[1..k]$ . Montrons-la par récurrence forte sur  $s$ .



#### Preuve

Cas négatif : Pour  $s < 0$ , la quantité de confiture est négative. Dans ce cas-là, on ne peut pas remplir les bocaux et on dit que le nombre de bocaux est infini. Par ailleurs, dans le corps de l'algorithme, on rentre dans le bloc `then` du premier branchement conditionnel et renvoie bien  $+\infty$ . De plus, on n'a effectué que des instructions élémentaires en nombre fini. De ces faits, `RechercheExhaustive` est valide et se termine.

Base : Pour  $s = 0$ , la quantité de confiture est nulle. Aussi, on rentre dans le bloc `else` du premier branchement conditionnel, puis dans le bloc `then` du deuxième branchement conditionnel. Donc, on renvoie 0, ce qui est correct puisque aucun bocal ne doit être rempli.

De la même manière que dans le cas où  $s < 0$ , on montre ici que l'algorithme se termine car la suite d'instructions effectuées ne comporte que des instructions élémentaires. De ces faits,  $P_1(0)$  est vraie.

Induction : Supposons  $P_1(s)$  vraie pour tout  $s < s_0$ , pour un  $s_0 > 0$  fixé. Montrons  $P_1(s_0)$  vraie.

Comme  $s_0 > 0$ , on rentre dans le bloc `else` du premier branchement conditionnel, puis dans le bloc `else` du deuxième branchement conditionnel.  

On ne rentre pas dans le branchement conditionnel. Donc, on renvoie le maximum de deux expressions :

- `tailleMaxRec`( $a, i + 1, j - 1$ ) +  $e(i, j)$ . Par hypothèse de récurrence,  $P(p)$  est vraie pour  $0 \leq p = j - i - 2 < p_0$ . De plus, d'après la remarque du cas négatif, la fonction se termine et est valide lorsque  $j - i - 2 < 0$ . Quant à  $e(i, j)$ , cette fonction ne contient que des *IE* et donc se termine. De plus, elle est bien valide car la valeur de retour est 0 ou 1 selon la possibilité de couplage entre  $i$  et  $j$ .
- $\max_{i < k \leq j} \text{tailleMaxRec}(a, i, k - 1) + \text{tailleMaxRec}(a, k, j)$ . Pour tout  $k \in \{i + 1, \dots, j\}$ ,
  - `tailleMaxRec`( $a, i, k - 1$ ) finit et est valide par hypothèse de récurrence :  $0 \leq k - 1 - i < p_0$ .

- `tailleMaxRec`( $a, k, j$ ) finit et est valide par hypothèse de récurrence :  $0 \leq j - k < p_0$ .

Par ailleurs, la fonction `max` est finie et renvoie bien le maximum des  $j - i$  valeurs de la somme.

Dans la deuxième partie de la question 3, on a montré une formule calculant  $E_{i,j}$ , c'est-à-dire la taille maximale d'une séquence de taille  $j - i + 1$  avec  $i < j$ , ce qui est bien le cas ici. On sait que cette formule est transcrite dans le code de la fonction `tailleMaxRec`. De plus, on vient de montrer que la valeur de retour de la fonction est finie et correspond au maximum des deux arguments décrits ci-dessus. De ces faits,  $P(p_0)$  est vraie.

Conclusion :

$$\left. \begin{array}{l} P_1(0) \text{ vraie} \\ \forall s_0 \geq 1, [(\forall s < s_0, P_1(s)) \Rightarrow P_1(s_0)] \end{array} \right\} \begin{array}{l} \forall s \in \mathbb{N}, \text{RechercheExhaustive}(k, V, s) \\ \text{se termine et est valide.} \end{array}$$

### Cas particulier

D'après le résultat précédant, pour  $s = S \in \mathbb{N}$  (la quantité de confiture dans le cadre de ce projet), on a que l'algorithme est valide et se termine lorsqu'il est appelé par l'appel initial `RechercheExhaustive`( $k, V, S$ ).

### Question 2

Soient  $b(s)$  et  $c(s)$  les suites définies par

$$b(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 & \text{si } s = 1 \\ 2 \cdot b(s-2) + 2 & \text{si } s \geq 2 \end{cases} \quad \text{et} \quad c(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 & \text{si } s = 1 \\ 2 \cdot c(s-1) + 2 & \text{si } s \geq 2 \end{cases}$$

Soit  $a(s)$  le nombre d'appels récursifs effectués par `RechercheExhaustive`(2, [1, 2],  $s$ ).

a) La suite  $a(s)$  est définie par

$$a(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 & \text{si } s = 1 \\ a(s-2) + a(s-1) + 2 & \text{si } s \geq 2 \end{cases}$$

b) Soit la propriété suivante  $P_2(s), s \geq 0 : b(s) \leq a(s) \leq c(s)$ . Montrons-la par récurrence d'ordre 2 sur  $s$ .

### Preuve

Base : Pour  $s = 0$ , on a  $b(0) = a(0) = c(0) = 0$ . Donc,  $P_2(0)$  est vraie.

Pour  $s = 1$ , on a  $b(1) = a(1) = c(1) = 2$ . Donc,  $P_2(1)$  est vraie.

Induction : Supposons  $P_2(s-2)$  et  $P_2(s-1)$  vraies pour un  $s \geq 2$  fixé. Montrons  $P_2(s)$  vraie.

Par hypothèses de récurrence, on a :

$$b(s-2) \leq a(s-2) \leq c(s-2) \quad (1)$$

$$b(s-1) \leq a(s-1) \leq c(s-1) \quad (2)$$

Par définition des suites récursives, on a :

$$b(s) = 2 \cdot b(s-2) + 2 \quad (3)$$

$$a(s) = a(s-2) + a(s-1) + 2 \quad (4)$$

$$c(s) = 2 \cdot c(s-1) + 2 \quad (5)$$

Étant donné que les suites  $b(s)$  et  $c(s)$  sont croissantes (c.f. annexe A), les équations (1) et (2) deviennent

$$b(s-2) \leq a(s-2) \leq c(s-1)$$

$$b(s-2) \leq a(s-1) \leq c(s-1)$$

Il en résulte que

$$2 \cdot b(s-2) + 2 \leq a(s-2) + a(s-1) + 2 \leq 2 \cdot c(s-1) + 2 \quad (6)$$

En remplaçant les définitions données en (3), (4) et (5) dans (6), on obtient  $b(s) \leq a(s) \leq c(s)$ .

Conclusion :

$$\left. \begin{array}{l} P_2(0) \text{ et } P_2(1) \text{ vraies} \\ \forall s \geq 2, [(P_2(s-2) \text{ et } P_2(s-1)) \Rightarrow P_2(s)] \end{array} \right\} \forall s \in \mathbb{N}, b(s) \leq a(s) \leq c(s)$$

- c) Soit la propriété suivante  $P_3(s)$ ,  $s \geq 0$  :  $c(s) = 2^{s+1} - 2$ . Montrons-la par récurrence faible sur  $s$ .

### Preuve

Base : Pour  $s = 0$ , on a  $c(0) = 0$ . De plus,  $2^{0+1} - 2 = 2^1 - 2 = 0$ . Donc,  $P_3(0)$  est vraie.

Induction : Supposons  $P_3(s)$  vraie pour un  $s \geq 0$  fixé. Montrons  $P_3(s+1)$  vraie.

On a  $s+1 \geq 1$ . Par définition,  $c(s+1) = 2 \cdot c(s) + 2$ . Or  $c(s) = 2^{s+1} - 2$ . Donc,  $c(s+1) = 2 \cdot (2^{s+1} - 2) + 2 = 2^{s+2} - 2$ . D'où,  $P_3(s+1)$  est vraie.

Conclusion :

$$\left. \begin{array}{l} P_3(0) \text{ vraie} \\ \forall s \geq 0, [P_3(s) \Rightarrow P_3(s+1)] \end{array} \right\} \forall s \in \mathbb{N}, c(s) = 2^{s+1} - 2$$

d)

Faire

e) On déduit des deux questions précédentes que  $b(s) = 2^{\lceil \frac{s}{2} \rceil + 1} - 2$ .

D'après la question b), on a  $b(s) \leq a(s) \leq c(s)$ .

De ces faits,  $2^{\lceil \frac{s}{2} \rceil + 1} - 2 \leq a(s) \leq 2^{s+1} - 2$ .

Considérons seulement les appels récurrents effectués par `RechercheExhaustive`(2, [1, 2], s) pour le calcul de la complexité temporelle de l'algorithme. Donc, la complexité pour le système des deux capacités minimales est donnée par  $a(s)$ . Ainsi, la complexité est exponentielle en  $S$ , à savoir la quantité de confiture à notre disposition.

### 3 Algorithme II : Programmation dynamique

#### Question 3

On note  $m(S)$  le nombre minimum de bords pour  $S$  décigrammes de confiture et un tableau d capacités  $V$ . On définit une famille de problèmes intermédiaires de la façon suivante : étant donné un entier  $s$  et un entier  $i \in \{1, \dots, k\}$ , on note  $m(s, i)$  le nombre minimum de bords nécessaires pour une quantité totale  $s$  en ne choisissant des bords que dans le système de capacités  $V[1], V[2], \dots, V[i]$ .

On pose :

$$\begin{aligned} m(0, i) &= 0 & \forall i \in \{1, \dots, k\} \\ m(s, 0) &= +\infty & \forall s \geq 1 \\ m(s, i) &= +\infty & \forall i \in \{1, \dots, k\} \forall s < 0 \end{aligned}$$

a) On a  $m(S) = m(S, k)$ .

b) Soit la relation de récurrence suivante pour tout  $i \in \{1, \dots, k\}$

$$m(s, i) = \begin{cases} 0 & \text{si } s = 0 \\ \min\{m(s, i-1), m(s - V[i], i) + 1\} & \text{sinon} \end{cases}$$

#### Preuve

Soit  $s$  la capacité de confiture à traiter. Deux cas sont possibles :

i)  $s = 0$  : Dans ce cas-là, il n'y a pas de confiture à préserver. On n'utilise donc aucun bocal, i.e. on a  $m(0, i) = 0$ . ■

ii)  $s \neq 0$  : Puisque  $m(s, i) = +\infty$  pour toute quantité  $s < 0$ , la relation de récurrence ne concerne dans ce cas-là que les quantités  $s > 0$ . De même, on a  $m(s, 0) = +\infty$  pour toute quantité  $s \geq 1$ . Donc, on ne considère que les  $i > 0$ .

Étant donné le système de capacités  $V[1], V[2], \dots, V[i]$ , deux choix sont possibles : soit on utilise des bords de capacité  $V[i]$ , soit on ne s'en sert pas.

D'un côté, le premier choix implique forcément l'utilisation d'*au moins* un bocal de ladite capacité. Ceci se traduit mathématiquement par  $m(s - V[i], i) + 1$ .

D'un autre côté, le deuxième choix rejete les bocaux de ladite capacité. Donc, le résultat est donné par  $m(s, i - 1)$ .

Comme on cherche à minimiser le nombre de bocaux à utiliser,  $m(s, i)$  doit valoir le minimum entre ces deux résultats. ■

## Question 4

Soit  $M$  un tableau doublement indicé tel que la case  $M[s, i]$ , où  $s \in \{0, \dots, S\}$  et  $i \in \{0, \dots, k\}$ , contient la valeur  $m(s, i)$ .

- a) Soit le couple  $(s, i) \in \{1, \dots, S\} \times \{1, \dots, k\}$ . D'après la relation de récurrence prouvée ci-dessus, pour calculer  $m(s, i)$ , on nécessite des valeurs stockées dans les cases  $(s, i-1)$  et  $(s-V[i], i)$ , si  $s-V[i] \geq 0$ . Autrement dit, la case à gauche et toutes les cases au-dessus de  $(s, i)$  doivent être traitées préalablement au calcul de la valeur de cette dernière.

Cela implique que le remplissage du tableau doit commencer à la case  $(0, 0)$  et aller jusqu'à la case  $(S, k)$  en suivant un parcours matriciel. Cependant, ce parcours peut se faire par lignes ou par colonnes.

**N.B.** Dans le cadre du projet, le parcours séquentiel des lignes a été sélectionné.

- b) On déduit des deux questions précédentes un algorithme **AlgoProgDyn** qui pour un système de  $k$  capacités  $V[1], V[2], \dots, V[k]$  et une quantité  $S$  de confiture retourne  $m(S)$ . En voici le pseudocode :

---

### Algorithm 1 AlgoProgDyn

---

**Require:**  $k \geq 0$  and  $S \geq 0$

```

1: function MAIN( $k$  : integer,  $V$  :  $k$ -integer array,  $S$  : integer)
2:    $M$  :  $S \times k$  integer matrix
3:   for  $s = 0$  to  $S$  do
4:     for  $i = 0$  to  $k$  do
5:        $M[s][i] \leftarrow \text{GETMSI}(V, M, s, i)$ 
6:   return  $M[S][k]$  ▷ The value of  $m(S)$ 

7: function GETMSI( $V$  :  $k$ -integer array,  $M$  :  $S \times k$  integer matrix,  $s$  : integer,
    $i$  : integer)
8:   if  $s = 0$  then
9:     return 0
10:  else if  $s < 0$  then
11:    return  $+\infty$ 
12:  else if  $i = 0$  then
13:    return  $+\infty$ 
14:  else
15:    return  $\min(\text{GETMSI}(V, M, s, i - 1), \text{GETMSI}(V, M, s - V[i], i) + 1)$ 

```

---



- c) Pour l'analyse de la complexité temporelle on utilise l'accès à une case du tableau  $M$  comme instruction élémentaire. Ainsi, on remarque que pour chaque case on accède au plus 3 fois à des éléments du tableau : l'une pour l'écriture de la case et les deux restantes pour la lecture des cases dont on cherche le minimum des valeurs. La complexité temporelle de l'algorithme est donc en  $\Theta(S \cdot k)$ .

Quant à la complexité spatiale, on se sert de deux structures de données gourmandes, à savoir le tableau  $V$  de  $k$  cases d'entiers et la matrice  $M$  de  $S \cdot k$  cases d'entiers. En prenant comme taille de base celle d'un entier, on réalise que la complexité spatiale de l'algorithme est aussi en  $\Theta(S \cdot k)$ .

## Question 5

On désire à présent permettre à l'algorithme de retourner un tableau  $A$  indiquant le nombre de boccas pris pour chaque type de capacités.

- a) Dans un premier temps, on décide de placer dans chaque case du tableau  $M[s, i]$ , un tableau  $A$  indiquant les boccas pris pour la capacité totale  $s$  et des capacités de boccas prises parmi  $V[1], \dots, V[i]$ .

Pour ce faire, on effectue quelques modifications dans l'algorithme **AlgoProgDyn**.

Tout d'abord, la matrice  $M$  devient une matrice de tableaux d'entiers d'ordre  $S \times k$ . Puis l'affectation de la case  $M[s][i]$  à la ligne 5 devient un appel à la procédure **SETMSI**( $V, M, s, i$ ). Cette dernière reprend la logique de la fonction **GETMSI**, sauf que l'on crée le tableau  $A$  pour la case correspondante au lieu renvoyer le nombre total de boccas utilisés.

Ce qui suit est faux

- b) Dans un deuxième temps, on souhaite éviter de copier dans chaque case du tableau  $M$ , les tableaux  $A$  des boccas utilisés. Pour ce faire, on conserve l'algorithme initial et on reconstitue *a posteriori* le tableau  $A$  de la solution optimale. Voici le pseudocode du deuxième algorithme :

Force est de constater que l'algorithme suit *dans le pire cas* un chemin de déplacements unitaires dans le même sens de la case  $(S, k)$  à la case  $(0, 0)$ . C'est pourquoi on effectue *au plus*  $S + k$  tours de boucle. On remarque aussi que le corps de la boucle comporte uniquement des instructions élémentaires. Ce morceau de l'algorithme est ainsi en  $\mathcal{O}(S + k)$ .

Par ailleurs, l'initialisation du tableau  $A$  est en  $\Theta(k)$ . On en déduit que la complexité de l'algorithme de retour est en  $\mathcal{O}(S + k)$ .

Complexité totale ?  
que le retour ?  
spatiale ?

## Question 6

Soit  $u_p$  le nombre d'appels de la fonction **tailleMaxRec** effectués pour  $p = j - i$ .

1.  $u_0 = 1$  car on rentre dans le branchement conditionnel.

$u_1 = 4$  car on ne rentre pas dans le branchement conditionnel et fait 1 appel pour le premier argument de la fonction **max** et 2 appels pour le deuxième argument.

---

```

1: function GETMSI( $V : k$ -integer array,  $M : S \times k$  integer array matrix,  $s :$ 
   integer,  $i : \text{integer}$ )
2:    $A : \text{integer array}$ 
3:   if  $s = 0$  then
4:     return 0
5:   else if  $s < 0$  then
6:     return  $+\infty$ 
7:   else if  $i = 0$  then
8:     return  $+\infty$ 
9:   else
10:     $A \leftarrow M[s][i]$ 
11:    return  $\sum_{j=1}^i A[j]$ 

12: function SETMSI( $V : k$ -integer array,  $M : S \times k$  integer array matrix,  $s :$ 
   integer,  $i : \text{integer}$ )
13:    $A : \text{integer array filled with } i \text{ zeros}$ 
14:   if  $s > 0$  and  $i > 0$  then
15:      $left \leftarrow \text{GETMSI}(V, M, s, i - 1)$ 
16:      $up \leftarrow \text{GETMSI}(V, M, s - V[i], i) + 1$ 
17:     if  $left < up$  then
18:        $A[1 \dots i - 1] \leftarrow M[s][i - 1]$ 
19:     else
20:        $A[1 \dots i] \leftarrow M[s - V[i]][i]$ 
21:        $A[i] \leftarrow A[i] + 1$ 
22:    $M[s][i] \leftarrow A$ 

```

---



---

**Algorithm 2** AlgoProgDynRet

---

**Require:**  $k \geq 0$  **and**  $S \geq 0$

---

```

1: function CONFIGUREBACKWARD( $k : \text{integer}$ ,  $V : k$ -integer array,  $S : \text{integer}$ ,
    $M : S \times k$  integer matrix)
2:    $A : \text{integer array filled with } k \text{ zeros}$ 
3:    $s, i : \text{integers}$ 
4:    $s \leftarrow S$ 
5:    $i \leftarrow k$ 
6:   while  $s \neq 0$  do
7:     if  $i > 0$  and  $M[s][i] = M[s][i - 1]$  then            $\triangleright$  no more capacity  $V[i]$ 
8:        $i \leftarrow i - 1$ 
9:     else                                                      $\triangleright M[s][i] = M[s - V[i]][i] + 1$ 
10:       $A[i] \leftarrow A[i] + 1$ 
11:       $s \leftarrow s - V[i]$ 
12:   return  $A$ 

```

---

2. Soit  $p \geq 2$ . Soit la propriété  $Q(p) : u_p = u_{p-2} + 1 + 2 \sum_{i=0}^{p-1} u_i$ .

Montrons-la par démonstration directe.

Soit  $p \geq 2$ . Comme  $p > 1$ , on ne rentre pas dans le branchement conditionnel de la fonction `tailleMaxRec`. Donc, on renvoie le maximum de deux expressions :

- `tailleMaxRec(a, i+1, j-1) + e(i, j)`. Ici, le nombre d'appels à la fonction est  $u_{p-2}$  car  $j - 1 - (i + 1) = j - i - 2 = p - 2$ .
- $\max_{i < k \leq j} \text{tailleMaxRec}(a, i, k - 1) + \text{tailleMaxRec}(a, k, j)$ . Pour tout  $k \in \{i + 1, \dots, j\}$ ,
  - `tailleMaxRec(a, i, k - 1)` réalise  $u_{k-1-i}$  appels à la fonction.
  - `tailleMaxRec(a, k, j)` réalise  $u_{j-k}$  appels à la fonction.

Donc, on a  $u_p = u_{p-2} + 1 + \sum_{k=i+1}^j u_{k-1-i} + u_{j-k}$ .

D'abord, on ré-indexe la somme :  $u_p = u_{p-2} + 1 + \sum_{k=0}^{j-i-1} u_k + u_{j-k-i-1}$ .

Or,  $p = j - i$ . Donc, on sépare les termes :

$$u_p = u_{p-2} + 1 + \sum_{k=0}^{p-1} u_k + \sum_{k=0}^{p-1} u_{p-1-k}.$$

Puis, on inverse l'indice de la deuxième somme :

$$u_p = u_{p-2} + 1 + \sum_{k=0}^{p-1} u_k + \sum_{k=0}^{p-1} u_k.$$

Finalement, on peut changer la variable des sommes et les regrouper :

$$u_p = u_{p-2} + 1 + 2 \sum_{i=0}^{p-1} u_i. \quad \square$$

3. Soit la propriété  $R(p) : 2^p \leq u_p \leq 4^p$ .

Montrons-la par récurrence forte sur  $p$ .

Base : Pour  $p = 0$ , d'après la sous-question 1,  $2^0 = 1 \leq u_0 = 1 \leq 1 = 4^0$ . Donc,  $R(0)$  est vraie.

Pour  $p = 1$ , d'après la sous-question 1,  $2^1 = 2 \leq u_1 = 4 \leq 4 = 4^1$ . Donc,  $R(1)$  est vraie.

Induction : Supposons  $R(p)$  vraie pour tout  $p < p_0$ , pour un  $p_0 > 1$  fixé. Montrons  $R(p_0)$  vraie.

D'après la sous-question 2,  $u_{p_0} = u_{p_0-2} + 1 + 2 \sum_{i=0}^{p_0-1} u_i$ .

Or, par hypothèse de récurrence, pour tout  $i \in \{0, \dots, p_0 - 1\}$ ,  $Q(i)$  est vraie. Donc,  $2^{p_0-2} + 1 + 2 \sum_{i=0}^{p_0-1} 2^i \leq u_{p_0} \leq 4^{p_0-2} + 1 + 2 \sum_{i=0}^{p_0-1} 4^i$ . D'où,

$$2^{p_0-2} + 1 + 2 \frac{2^{p_0} - 1}{2 - 1} \leq u_{p_0} \leq 4^{p_0-2} + 1 + 2 \frac{4^{p_0} - 1}{4 - 1}, \text{ c'est-à-dire}$$

$$2^{p_0} \leq 2^{p_0-2} + 2^{p_0+1} - 1 \leq u_{p_0} \leq 4^{p_0-2} + 2 \frac{4^{p_0}}{3} + \frac{1}{3} \leq 4^{p_0}.$$

Conclusion :

$$\left. \begin{array}{l} R(0), R(1) \text{ vraies} \\ \forall p_0 \in \{2, \dots, n-1\}, [(\forall p < p_0, R(p)) \implies R(p_0)] \end{array} \right\} \begin{array}{l} \forall p \in \{0, \dots, n-1\}, \\ 2^p \leq u_p \leq 4^p \end{array}$$

4. D'une part, d'après le paragraphe précédent,  $R(n-1)$  est vraie, c'est-à-dire  $2^{n-1} \leq u_{n-1} \leq 4^{n-1}$ , où  $n = p+1 = j-i+1$  est la longueur de la séquence.

D'autre part, chaque appel à `tailleMaxRec` ne contient que des *IE* à part ses appels récursifs (on considère que la fonction `max` est optimale et fait donc les comparaisons *in situ*).

Donc, on peut exprimer la complexité de la fonction en nombre d'appels récursifs effectués. De ce fait, la complexité de `tailleMaxRec` est en  $\Omega(2^n)$  et  $\mathcal{O}(4^n)$ .

## 4 Algorithme III : Cas particulier et algorithme glouton

### Question 7

On utilisera un tableau à deux dimensions pour stocker les valeurs de  $E_{i,j}$ . De plus, on considérera que l'accès à une case du tableau et l'appel à `e()` sont des instructions élémentaires compte tenu de leur complexité :  $\Theta(1)$ .

Tout d'abord, on initialise la première case de la première ligne à 0. Donc, la complexité de cette partie est en  $\Theta(1)$ .

Puis, la première boucle `for` est effectuée  $n-1$  fois et à chaque tour de boucle on ne réalise que des *IE*. Donc, la complexité de cette partie est en  $\Theta(n)$ .

Finalement, on effectue deux boucles l'une à l'intérieur de l'autre. La boucle imbriquée consiste en le calcul du maximum de deux expressions. Les opérations arithmétiques n'étant pas d'instructions représentatives, la première d'entre elles comprend deux *IE*. En ce qui concerne le deuxième argument, le nombre d'*IE* effectuées est  $2p$ . Donc, le nombre d'*IE* réalisées dans cette partie est

$$c = \sum_{p=1}^{n-1} \sum_{i=1}^{n-p} 2p + 2 = \sum_{p=1}^{n-1} (2p+2)(n-p) = 2 \sum_{p=1}^{n-1} [p(n-1) + n - p^2].$$

D'où,  $c = (n-1)^2n + 2(n-1)n - \frac{(n-1)n(2n-1)}{3} = \frac{(n-1)n(n+4)}{3}$ . Donc, la complexité de cette partie est en  $\Theta(n^3)$ .

De ces trois faits, on en déduit que la complexité de l'algorithme est en  $\Theta(n^3)$ .

## Deuxième partie

# Mise en œuvre

## 5 Implémentation

Le but de ce second exercice est de mesurer de manière expérimentale la complexité de la fonction `tailleMaxRec` et de l'algorithme décrit dans la question 5 de l'exercice précédent. Le choix du langage de programmation est libre. La complexité expérimentale de l'exécution d'une fonction correspond ici au temps qu'il faut pour que la fonction s'exécute.

Dans cet exercice,  $n$  désigne la taille de la séquence initiale.

### Question 1

Voici le code des fonctions `e`, `maximum`, `tailleMaxRec` et `tailleMaxIter`.

```
def e(a,i,j):
    """ str x int x int -> bool
    rend True si i et j peuvent être couplés dans a, False
    sinon
    """
    # couples : list[tuple[str]]
    couples = [('A','T'),('A','T'),('G','C'),('C','G')]
    if (a[i-1],a[j-1]) in couples:
        return True
    return False

def maximum(a,i,j):
    """ str x int x int -> int
    rend le nombre de couples de la structure secondaire de
    taille maximale obtenue pour l'un des partitionnements
    de la sous-séquence  $a_{i,j}$ 
    """
    # maxi : int
    maxi = 0
    # k : int
    for k in range(i,j):
        # val : int
        val = tailleMaxRec(a,i-1,k-1)+tailleMaxRec(a,k,j-1)
        if val > maxi:
            maxi = val
    return maxi

def tailleMaxRec(a,i,j):
    """ str x int x int -> int
    rend le nombre de couples de la structure secondaire de
    taille maximale de la sous-séquence  $a_{i,j}$ 
    """
    if j-i < 1:
        return 0
```

```

return max(tailleMaxRec(a,i+1,j-1) + e(a,i,j), \
           maximum(a,i+1,j+1))

def tailleMaxIter(a,i,j):
    """ str x int x int -> list[list[int]]
    rend un tableau  $\tilde{A}$  2 dimensions contenant le nombre de
    couples de la structure secondaire de taille maximale de
    chaque sous-sequence  $a_{k,l}$  avec  $i \leq k, l \leq j$  et  $k \leq l$ 
    """
    # n : int
    n = len(a)
    # E : list[list[int]]
    E = [[0]]
    # i : int
    for i in range(2,n+1,1):
        E.append([0]*i)
    # p : int
    for p in range(1,n):
        for i in range(1,n-p+1):
            E[i-1].append(max(E[i][i+p-2]+e(a,i,i+p), \
                               max(E[i-1][k-2]+E[k-1][i+p-1] \
                                   for k in range(i+1,i+p+1))))
    return E[i-1][j-1]

```

Voici un jeu d'essai.

## Question 2

Voici le code de la fonction `SeqAleatoire(n)` qui renvoie une séquence d'ADN aléatoire de taille  $n$ .

```
import random

def SeqAleatoire(n):
    """ int -> str
    rend une séquence aléatoire d'ADN de taille n
    """
    seq = ''
    for i in range(n):
        seq += random.choice('AGCT')
    return seq
```

## Question 3

La plus grande valeur de  $n$  que l'on peut traiter sans problème de mémoire ou de temps d'exécution de quelques minutes pour la fonction `tailleMaxRec` est 18.

Quant à `tailleMaxIter`, la valeur de  $n$  correspondante est 1600.

## Question 4

On a mis dans le tableau ci-dessous le temps d'exécution de plusieurs appels à la fonction `tailleMaxRec` selon les valeurs de  $n$ .

$n$	$CRec(n)$ (en s.)	$\log CRec(n)$
11	0.0915572	-1.0383075
12	0.3087260	-0.5104268
13	0.9356748	-0.0288751
14	3.0208589	0.4801304
15	9.7992974	0.9911949
16	30.8693360	1.4895273
17	100.5011405	2.0021710
18	322.5870859	2.5086470

On en déduit la droite de régression de  $\log CRec(n)$  en fonction de  $n$  :  
 $\Delta : y = 0.505452x - 6.592292$ . On remarque que la pente de la droite est 0.505452, ce qui était attendu car la valeur est comprise entre  $\log 2 = 0.301030$  et  $\log 4 = 0.602060$ .

## Question 5

On a mis dans le tableau ci-dessous le temps d'exécution de plusieurs appels à la fonction `tailleMaxIter` selon les valeurs de  $n$ .

$n$	$CIter(n)$ (en s.)	$\frac{CIter(n)}{n^3}$
200	0.5192115	$6.490\,143\,4 \times 10^{-8}$
400	4.1114257	$6.424\,102\,6 \times 10^{-8}$
600	14.2562266	$6.600\,104\,9 \times 10^{-8}$
800	34.8566699	$6.807\,943\,3 \times 10^{-8}$
1000	71.8084174	$7.180\,841\,7 \times 10^{-8}$
1200	124.3441816	$7.195\,843\,8 \times 10^{-8}$
1400	200.7089246	$7.314\,465\,2 \times 10^{-8}$
1600	314.0727653	$7.667\,792\,1 \times 10^{-8}$

On en déduit la droite de régression de  $\frac{CIter(n)}{n^3}$  en fonction de  $n$  :  
 $\Delta : y = 8.842\,541 \times 10^{-12}x - 6.164\,326 \times 10^{-8}$ . On remarque l'ordre de grandeur de la pente de la droite :  $8.842\,541 \times 10^{-12} \ll 1$ . Ceci implique que la fonction est bien constante de valeur  $6.960\,154\,6 \times 10^{-8}$ .



## A Suites croissantes

On rappelle la définition des suites  $b(s)$  et  $c(s)$  comme suit :

$$b(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 & \text{si } s = 1 \\ 2 \cdot b(s-2) + 2 & \text{si } s \geq 2 \end{cases} \quad \text{et} \quad c(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 \cdot c(s-1) + 2 & \text{si } s \geq 1 \end{cases}$$

Montrons que les deux suites sont croissantes.

### Suite $b(s)$

Soit la propriété suivante  $Q_1(s), s \geq 0 : b(s) \leq b(s+1)$ . Montrons-la par récurrence d'ordre 2 sur  $s$

#### Preuve

Base : On a  $b(0) = 0 \leq 2 = b(1) \leq 2 \cdot 0 + 2 = b(2)$ . Donc, la propriété est vérifiée aux rangs 0 et 1.

Induction : Supposons  $Q_1(s-2)$  et  $Q_1(s-1)$  vraies pour un  $s \geq 2$  fixé. Montrons  $Q_1(s)$  vraie.

Par hypothèse de récurrence,  $b(s-2) \leq b(s-1)$ . D'où,  $2 \cdot b(s-2) + 2 \leq 2 \cdot b(s-1) + 2$ . Or, par définition,  $b(s+1) = 2 \cdot b(s-1) + 2$ . Donc,  $b(s) \leq b(s+1)$ , i.e. la propriété est vérifiée au rang  $s$ .

Conclusion :

$$\left. \begin{array}{l} Q_1(0) \text{ et } Q_1(1) \text{ vraies} \\ \forall s \geq 2, [(Q_1(s-2) \text{ et } Q_1(s-1)) \Rightarrow Q_1(s)] \end{array} \right\} \quad \forall s \in \mathbb{N}, b(s) \leq b(s+1)$$

On remarque que ceci correspond à la définition d'une suite croissante. ■

### Suite $c(s)$

Soit la propriété suivante  $Q_2(s), s \geq 0 : c(s) \geq 0$ . Montrons-la par récurrence faible sur  $s$ .

#### Preuve

Base : Pour  $s = 0$ , on a  $c(0) = 0 \geq 0$ . Donc, la propriété est vérifiée au rang 0.

Induction : Supposons  $Q_2(s)$  vraie pour un  $s \geq 0$  fixé. Montrons  $Q_2(s+1)$  vraie.

Par définition,  $c(s+1) = 2 \cdot c(s) + 2$ . Or, par hypothèse de récurrence,  $c(s) \geq 0$ . D'où,  $2 \cdot c(s) + 2 \geq 2$ . Donc,  $c(s+1) \geq 2 \geq 0$ . On en déduit par transitivité que la propriété est vérifiée au rang  $s+1$ .

Conclusion :

$$\left. \begin{array}{l} Q_2(0) \text{ vraie} \\ \forall s \geq 0, [Q_2(s) \Rightarrow Q_2(s+1)] \end{array} \right\} \forall s \in \mathbb{N}, c(s) \geq 0$$

Par ailleurs, on a  $c(s) = 2 \cdot c(s-1) + 2, s \geq 1$ . D'où,  $c(s) \geq 2 \cdot c(s-1)$ . Comme cette suite est positive, on sait que  $2 \cdot c(s-1) \geq c(s)$ . Donc, par transitivité,  $c(s) \geq c(s-1)$ . Autrement dit, pour tout  $s \geq 0, c(s+1) \geq c(s)$ . ■

## B Complexité de Recherche Exhaustive dans le cas général