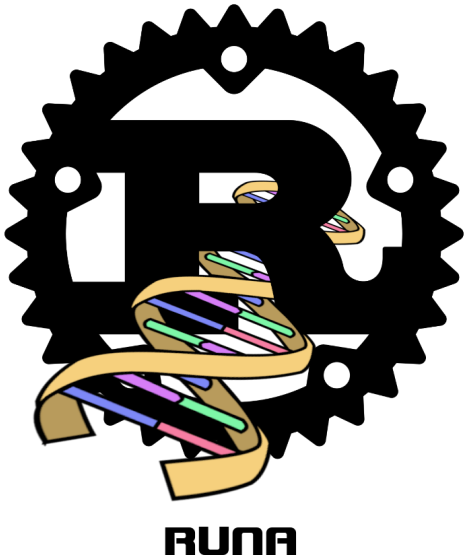
RuNA : Alignement de séquences

Rodrigue Tavernier, Kacper Ozieblowski

September 2022



1 Introduction

Toutes les commandes nécessaires pour exécuter les tests et les différentes fonctions sont disponibles dans le README.

2 Le problème d'alignement de séquences

Question 1 :

Soient $(\overline{x}, \overline{y})$ et $(\overline{u}, \overline{v})$ des alignements de (x, y) et (u, v). Montrer que $(\overline{x} \cdot \overline{u}, \overline{y} \cdot \overline{v})$ est un alignement de $(x \cdot u, y \cdot v)$.

1. On a la propriété suivante $\pi(\overline{x} \cdot \overline{u}) = \pi(\overline{x}) \cdot \pi(\overline{u})$. En effet, π ne change pas l'ordre des caractères donc l'ordre est conservé dans le résultat. Ainsi $\pi(\overline{x} \cdot \overline{u}) = \pi(\overline{x}) \cdot \pi(\overline{u}) = x \cdot u$.
2. De même $\pi(\overline{y} \cdot \overline{v}) = y \cdot v$.
3. Comme $|\overline{x}| = |\overline{y}|$ et $|\overline{u}| = |\overline{v}|$ par hypothèse, on a directement par concaténation $|\overline{x} \cdot \overline{u}| = |\overline{x}| + |\overline{u}| = |\overline{y}| + |\overline{v}| = |\overline{y} \cdot \overline{v}|$.
4. Comme $\forall i \in [1 \dots |\overline{x}|], \overline{x_i} \neq -$ ou $\overline{y_i} \neq -$ et $\forall j \in [1 \dots |\overline{u}|], \overline{u_j} \neq -$ ou $\overline{v_j} \neq -$. Alors $\forall k \in [1 \dots |\overline{x} \cdot \overline{u}|], (\overline{x} \cdot \overline{u})_k \neq -$ ou $(\overline{y} \cdot \overline{v})_k \neq -$. Ce qui termine la preuve.

Question 2 :

Soient x et y deux mots de longueurs n et m. Trouver la longueur maximale d'un alignement.

On peut construire un alignement où aucun caractère de x ne fait face à un caractère de y (ie chaque caractère est associé à un gap). Ainsi la longueur maximale est $n + m$. Voici un exemple :

$$x_1 \dots x_n - - - -$$
$$- - - - y_1 \dots y_m$$

3 Algorithmes pour l'alignement de séquences

Question 3 :

Étant donné $x \in \Sigma^*$, un mot de longueur n, combien y a-t-il de mots $\overline{x}$ obtenus en ajoutant à x exactement k gaps ?

On peut voir $\overline{x}$ comme un tableau à $n + k$ cases. Dans ce cas, le problème revient à placer k gaps dans $n + k$ cases. Il y a alors $\binom{n+k}{k}$ façons de construire ce mot.

Question 4 :

On cherche maintenant à en déduire le nombre d'alignements possibles d'un couple de mots (x, y) de longueurs respectives n et m, en supposant que $n \geq m$. Une fois ajoutés k gaps à x pour obtenir un mot $\overline{x} \in \overline{\Sigma}^*$, combien de gaps seront ajoutés à y ? Combien y a-t-il de façons d'insérer ces gaps dans y sachant qu'un gap du mot $\overline{y} \in \overline{\Sigma}^*$ ainsi obtenu ne doit pas être placé à la même position qu'un gap de $\overline{x}$? En déduire le nombre d'alignements possibles de (x, y).
Donner le nombre d'alignements possibles pour $|x| = 15$ et $|y| = 10$.

1. $|\overline{x}| = n + k$ et $|y| = m$ donc comme $|\overline{x}| = |\overline{y}|$ alors $|gap(\overline{y})| = |\overline{x}| - |y| = n + k - m$.

2. Il y a k gap dans $\overline{x}$ donc il y a k lettres de y associées à ces gaps dans $\overline{y}$. Il reste donc $m - k$ lettres de y à placer parmi les n lettres de $\overline{x}$. Cela donne $\binom{n}{m-k}$ possibilités.

3. Le nombre d'alignements possibles (pour un k donné) est le nombre de façons de placer les k gaps dans $\overline{x}$ multiplié par le nombre de façon de d'insérer les $n + k - m$ gaps dans $\overline{y}$.

   En utilisant le fait que $m \geq k$ (sinon on pourrait avoir 2 gaps ensemble), on sait qu'on peut ajouter au plus m gaps dans x. On donc $m+1$ possibilités pour k. Ainsi $|alignements(x, y)| = \sum_{k=0}^{m} \binom{n}{k} \times \binom{n}{m-k}$.

   Pour $|x| = 15$ et $|y| = 10$ le nombre d'alignements possibles est 30045015.

Question 5 :

Quel genre de complexité temporelle aurait un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ? En vue de trouver un alignement de coût minimal ?

Pour trouver la distance d'édition, on parcours tous les alignements possibles, pour chaque alignement on calcule son coût. On retient le plus petit de ces coûts. Le calcul du coût se fait en $O(n)$ la taille du mot, et on fait ce calcul pour chaque alignement soit $\sum_{k=0}^{m} \binom{n}{k} \times \binom{n}{m-k}$ fois. Ce qui donne une complexité en $O(n \times \sum_{k=0}^{m} \binom{n}{k} \times \binom{n}{m-k})$.
Trouver l'alignement de coût minimal se fait par la même démarche, mais en retenant l'alignement en plus.

Question 6 :

Quelle complexité spatiale (ordre de grandeur de la place requise en mémoire) aurait un algorithme naïf qui consisterait à énumérer tous les alignements de deux mots en vue de trouver la distance d'édition entre ces deux mots ? en vue de trouver un alignement de coût minimal ?

Puisque la fonction alloue une quantité constante de mémoire supplémentaire a chaque appel, la complexité mémoire dépend uniquement de la taille de la pile d'appels. Ainsi elle dépend grandement de la complexité en terme de nombre d'appels de notre fonction :

```
pub fn dist_naif_rec<T, I>(mut xi: I, mut yi: I, c: T::Cost, mut dist: T::Cost)
-> T::Cost
where T: MetricSpace, I: Iterator<Item = T::Item> + Clone
{
    let (xo, yo) = (xi.clone(), yi.clone());
    let m = (xi.next(), yi.next());
    if let (None, None) = m { return c.min(dist); }
    if let (Some(xj), Some(yj)) = m {
        dist = dist_naif_rec::<T, I>(xi.clone(), yi.clone(),
            c + T::sub(xj, yj), dist);
    }
    if let (Some(_), _) = m {
        dist = dist_naif_rec::<T, I>(xi, yo, c + T::DEL, dist);
    }
    if let (_, Some(_)) = m {
        dist = dist_naif_rec::<T, I>(xo, yi, c + T::INS, dist);
    }
    dist
}
```

On voit bien que dans le pire cas on fait 3 appels supplémentaires a chaque itération. Notre complexité mémoire est ainsi exponentielle en fonction de la taille du plus grand mot. De la même manière, une fonction naïve qui cherche a trouver un alignement optimal alloue une quantité mémoire qui est constante à chaque appel si on ignore les appels récursifs. En effet, la fonction ne fait que copier des pointeurs. Sa complexité est donc également proportionelle a la taille de la pile d'appels. Elle est donc exponentielle.

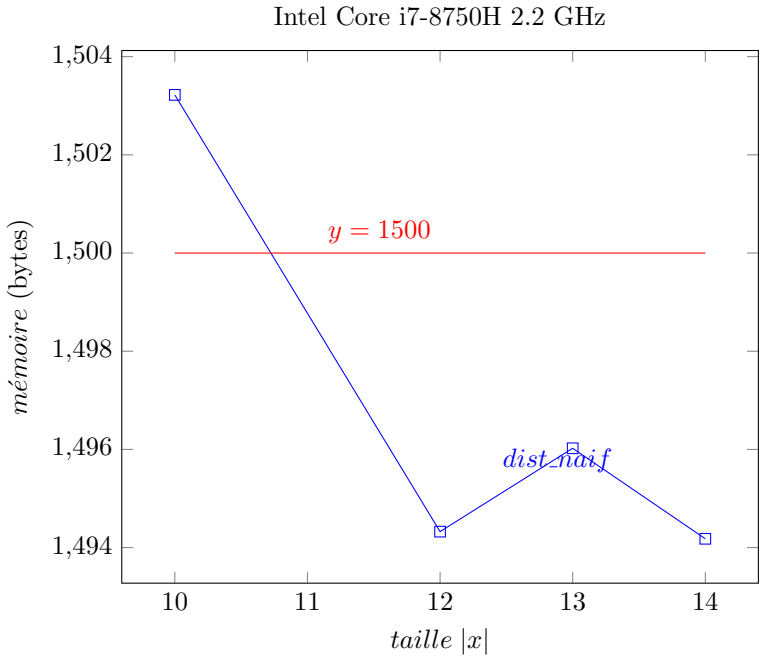
Question tache A :

Coder les fonctions **DIST_NAIF** et **DIST_NAIF_REC**.

- Tester la validité de votre implémentation sur les instances Inst_0000010_44.adn, Inst_0000010_7.adn et Inst_0000010_8.adn qui ont pour distance d'édition 10, 8 et 2.
- Pour évaluer les performances de cette méthode, évaluez jusqu'à quelle taille d'instance vous pouvez résoudre les instances fournies en moins d'une minute.
- Estimer la consommation mémoire nécessaire au fonctionnement de cette méthode. Dans le cas d'une consommation mémoire qui est réalisée uniquement au début d'un programme, cela peut-être fait simplement en lançant la commande Linux top pendant le début de l'exécution (l'affichage de top est mis à jour en appuyant sur espace). Il se peut aussi que la consommation mémoire demandée soit trop importante et que le système stoppe alors le programme.

- Nous remarquons que la taille d'instance maximum pour laquelle on peut résoudre les instances fournies en moins d'une minute est 14.
- Pour meusuer la consommation mémoire nous utilisons valgrind. Plus particulièrement nous utilisons la commande suivante : **GENOME_DATA=./tests/genome_instances_data/ valgrind –tool=massif ./target/debug/deps/mem_use-67d566998f108632 dist_naif** Le main utilisé se trouve dans le fichier tests/mem_use.rs
- La mémoire utilisée sur le tas par la fonction **DIST_NAIF** est de 1736 bytes, soit 1.7KB pour l'instance Inst_0000014_83.adn.

  Nous n'avons pas pu mesurer l'utilisation de la mémoire sur la pile. Or la place mémoire utilisée pour **DIST_NAIF** se trouve principalement sur la pile. En revanche nous avons essayé d'estimer grossièrement la mémoire utilisée sur le tas, même si cela n'est pas représentatif de la mémoire totale utilisée. En voici le graphe :



4 Programmation dynamique

Question 7 :

Soit $(\overline{u}, \overline{v})$ un alignement de $(x_{[1..i]}, y_{[1..j]})$ de longueur l. Si Si $\overline{u}_l = -$, que vaut $\overline{v}_l$? Si $\overline{v}_l = -$, que vaut $\overline{u}_l$? Si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$, que valent $\overline{u}_l$ et $\overline{v}_l$?

1. Si $\overline{u}_l = -$ on a $\overline{v}_l = v_j$ car le dernier élément, s'il n'est pas un *gap*, est forcément le dernier élément du mot.
2. Si $\overline{v}_l = -$ on a $\overline{u}_l = u_i$ pour la même raison.
3. Si $\overline{u}_l \neq -$ et $\overline{v}_l \neq -$ on a $\overline{u}_l = x_i$ et $\overline{v}_l = y_j$ car le dernier élément des 2 mots de l'alignement n'est pas un *gap*. Or le dernier caractère(qui n'est pas un *gap*) d'un alignement est forcément le dernier caractère du mot de base.

Question 8 :

En distinguant les trois cas envisagés à la question 7, exprimer $C(\overline{u}, \overline{v})$ à partir de $C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]})$

En reprenant la question 7 :

$$C(\overline{u}, \overline{v}) = \begin{cases} c_{del} + C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) & \text{si } \overline{u}_l = - \\ c_{ins} + C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) & \text{si } \overline{v}_l = - \\ c_{sub}(\overline{u}_l, \overline{v}_l) + C(\overline{u}_{[1..l-1]}, \overline{v}_{[1..l-1]}) & \text{sinon} \end{cases}$$

Question 9 :

Pour $i \in [1..n]$ et $j \in [1..m]$, déduire des questions 7 et 8 l'expression de $D(i, j)$ à partir des valeurs de D à des rangs plus petits.

Peut importe l'alignement qu'on considère, il tombe forcément dans l'un des cas des questions précédentes. On a:

$$D(i,j) = min\{C(\overline{x},\overline{y})\}$$

où $(\overline{x},\overline{y})$ est un alignement de $(x_{[1..i]}, y_{[1..j]})$. Ainsi, pour trouver le coût minimum on procède récursivement. On a pour $i > 0$ et $j > 0$:

$$D(i,j) = \min \begin{cases} D(i,j-1) + c_{ins} \\ D(i-1,j) + c_{del} \\ D(i-1,j-1) + c_{sub}(x_i, y_j) \end{cases}$$

Pour les cas "$i > 0$ et $j = 0$" et "$i = 0$ et $j > 0$" on utilise une restriction des cas seulement. L'expression ne fait pas intervenir les alignements directement car il ne sont pas nécessaires pour déterminer le coût minimum.

Question 10 :

Que vaut $D(0,0)$?

On considère le mot de longueur 0 comme le mot vide ε. Donc cela revient à comparer la distance d'édition entre 2 mots vides. On obtient $D(0,0) = 0$.

Question 11 :

Que vaut $D(0,j)$ pour $j \in [1..m]$? Que vaut $D(i,0)$ pour $i \in [1..n]$?

Cela revient à trouver la distance entre le mot vide et $x_{[1..i]}$ ou $y_{[1..j]}$. Si on veut un alignement de $(x_{[1..i]}, \varepsilon)$, on aura i *gaps* dans $\overline{\varepsilon}$ et $x_{[1..i]} = \overline{x_{[1..i]}}$. Ainsi $D(i,0) = i \times c_{del}$.
On raisonne de la même manière pour $y_{[1..j]}$ et on a alors $D(0,j) = j \times c_{ins}$.

Question 12 :

Donner le pseudo-code d'un algorithme itératif nommé **DIST_1**, qui prend en entrée deux mots, qui remplit un tableau à deux dimensions T avec toutes les valeurs de D pour finalement renvoyer la distance d'édition entre ces deux mots.

Procedure 1 DIST_1

Input: x, y

$n \leftarrow |x| + 1$
$m \leftarrow |y| + 1$
T tableau de taille $n \times m$
$T[0][0] \leftarrow 0$
for i allant de 1 à n (exclu) **do**
  $T[i][0] \leftarrow T[i-1][0] + c_{ins}$
end for
for j allant de 1 à m (exclu) **do**
  $T[0][j] \leftarrow T[0][j-1] + c_{del}$
end for
for i allant de 1 à n (exclu) **do**
  for j allant de 1 à m (exclu) **do**

$$T[i][j] \leftarrow \min\begin{pmatrix} T[i-1][j-1] + c_{sub}(x_{i-1}, y_{j-1}), \\ T[i][j-1] + c_{ins}, \\ T[i-1][j] + c_{del} \end{pmatrix}$$

  end for
end for

Output: $T[|x|][|y|]$

Question 13 :

Quelle est la complexité spatiale de **DIST_1**?

Soit x, y deux mots de tailles n et m respectivement. Le tableau T contient $n \times m$ entiers. Les autres variables sont de taille constante. Donc la complexité spatiale est de $O(n \times m)$

Question 14 :

Quelle est la complexité temporelle de **DIST_1**?

Il y a deux boucles imbriquées, donc $n \times m$ itérations. Chaque itération se fait en $O(1)$ (opérations élémentaires et accès au tableau). Donc la complexité de cette boucle est en $O(m \times n)$. Le coûts des deux autres boucles sont en $O(n)$ et $O(m)$ respectivement. Il sont donc négligeable par rapport au $O(n \times m)$. Ainsi la complexité finale est en $O(m \times n)$.

Question 15 :

Soit $(i, j) \in [0..n] \times [0..m]$. Montrer que :

1. Si $j > 0$ et $D(i, j) = D(i, j-1) + c_{ins}$ alors $\forall (\overline{s}, \overline{t}) \in Al^*(i, j-1)$, $(\overline{s} \cdot -, \overline{t} \cdot y_j) \in Al^*(i, j)$
2. Si $i > 0$ et $D(i, j) = D(i-1, j) + c_{del}$ alors $\forall (\overline{s}, \overline{t}) \in Al^*(i-1, j)$, $(\overline{s} \cdot x_i, \overline{t} \cdot -) \in Al^*(i, j)$
3. $D(i, j) = D(i-1, j-1) + c_{sub}(x_i, x_j)$ alors $\forall (\overline{s}, \overline{t}) \in Al^*(i-1, j-1)$, $(\overline{s} \cdot x_i, \overline{t} \cdot y_j) \in Al^*(i, j)$

Soit $(\overline{s}, \overline{t}) \in Al^*(i, j-1)$ un alignement de coût minimum de $(s_{[1..i]}, t_{[1..j-1]})$. Par hypothèse $C(\overline{s}_{[1..i]}, \overline{t}_{[1..j]})$ est minimum si partant d'un alignement de $(s_{[1..i]}, t_{[1..j-1]})$, on fait une insertion supplémentaire, c'est à dire, si on rajoute un *gap* dans $\overline{s_{[1..i]}}$ à la même position(à la fin dans notre cas) que le caractère qu'on ajoute dans $\overline{t_{[1..j-1]}}$ (par définition de l'insertion). Cela signifie donc que $(\overline{s_{[1..i]}} \cdot -, \overline{t_{[1..j-1]}} \cdot y_j)$ est un alignement, et qu'il est de coût minimum. On remarque facilement que $\pi(\overline{t_{[1..j-1]}} \cdot y_j) = y_{[1..j]}$ donc l'alignement obtenu appartient bien à $Al^*(i, j)$. Les autres cas sont

analogues.

Question 16 :

Donner le pseudo-code d'un algorithme itératif nommé **SOL_1**, qui à partir d'un couple de mots (x, y) et d'un tableau T indexé par $[0..|x|]\times[0..|y|]$ contenant les valeurs de D, renvoie un alignement minimal de (x, y).

Procedure 2 SOL_1

Input: x, y, $T[0..|x|] \times [0..|y|]$

```
i ← |x|
j ← |y|
x̄ ← []
ȳ ← []
while i > 0 et j > 0 do
    if T[i][j] = T[i − 1][j − 1] + c_sub(x_{i−1}, y_{j−1}) then
        x̄ ← x̄ · x_{i−1}
        ȳ ← ȳ · y_{j−1}
        i ← i − 1
        j ← j − 1
    else if T[i][j] = T[i][j − 1] + c_ins then
        x̄ ← x̄ · −
        ȳ ← ȳ · y_{j−1}
        j ← j − 1
    else
        x̄ ← x̄ · x_{i−1}
        ȳ ← ȳ · −
        i ← i − 1
    end if
end while
while i > 0 do
    x̄ ← x̄ · x_{i−1}
    ȳ ← ȳ · −
    i ← i − 1
end while
while j > 0 do
    x̄ ← x̄ · −
    ȳ ← ȳ · y_{j−1}
    j ← j − 1
end while
reverse(x̄)
reverse(ȳ)
```

Output: $(\overline{x}, \overline{y})$

Question 17 :

En combinant les algorithmes **DIST_1** et **SOL_1** avec quelle complexité temporelle résout-on le problème ALI?

La complexité temporelle de **SOL_1** est en $O(n+m)$. En effet, le nombre d'itérations de chacune des trois boucles while de la procédure **SOL_1** est inférieur a $n + m$. Les opérations a l'intérieur de ces

boucles sont en $O(1)$. Donc en combinant les deux algorithmes on a une complexité $O(n \times m + n + m) = O(n \times m)$

Question 18 :

En combinant les algorithmes **DIST_1** et **SOL_1** avec quelle complexité spatiale résout-on le problème ALI?

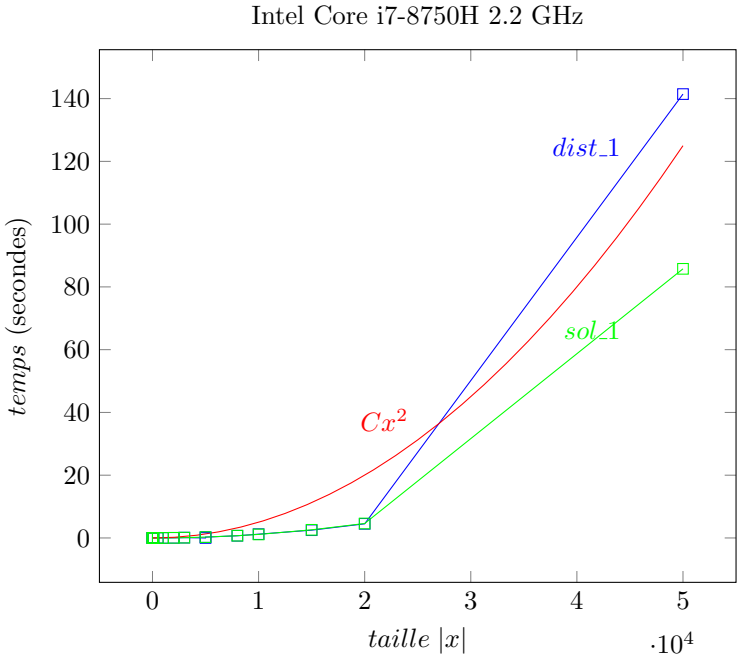
La complexité spatiale de **SOL_1** est en $O(n + m)$. En effet, par la question 2, la taille maximum d'un alignement est de $n + m$. Les autres variables dans **SOL_1** sont de taille constante. Ainsi on résout le problème ALI en complexité spatiale de $O(n + m + n \times m) = O(n \times m)$

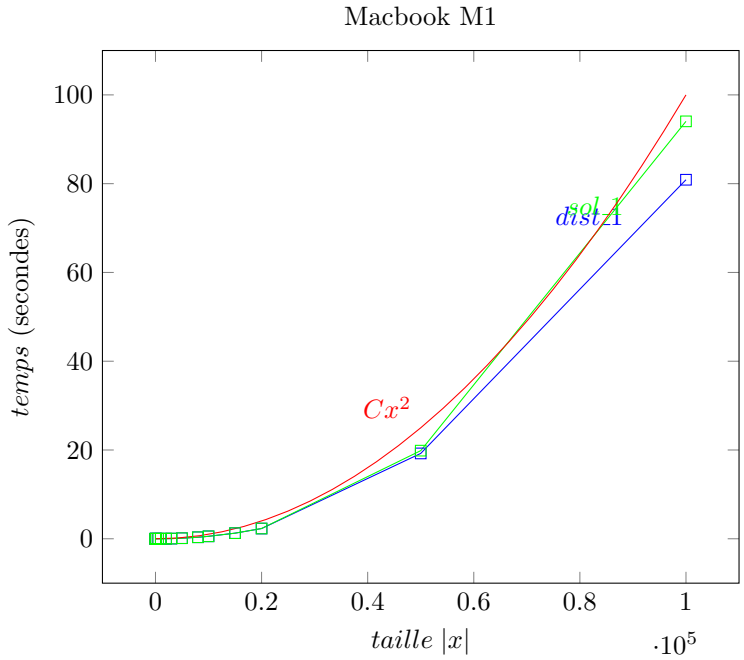
Question tache B :

- Coder les deux fonctions **DIST_1** et **SOL_1**, ainsi qu'une fonction **PROG_DYN** qui prend en entrée seulement les mots x et y et qui renvoie à la fois la distance $d(x, y)$ et un alignement optimal. Tester ces fonctions sur plusieurs instances.
- Tracer la courbe de consommation de temps CPU en fonction de la taille $|x|$ du premier mot du couple des instances fournies. Est-ce que la courbe obtenue correspond à la complexité théorique ? Vous pouvez vous limiter aux instances nécessitant moins de 10 minutes chacune.
- Estimer la quantité de mémoire utilisée par **PROG_DYN** pour une instance de très grande taille.

Pour lancer les tests puis pour obtenir les performances on utilise les commandes:

```
$ cargo test
$ cargo bench -- gnuplot dist_1
```

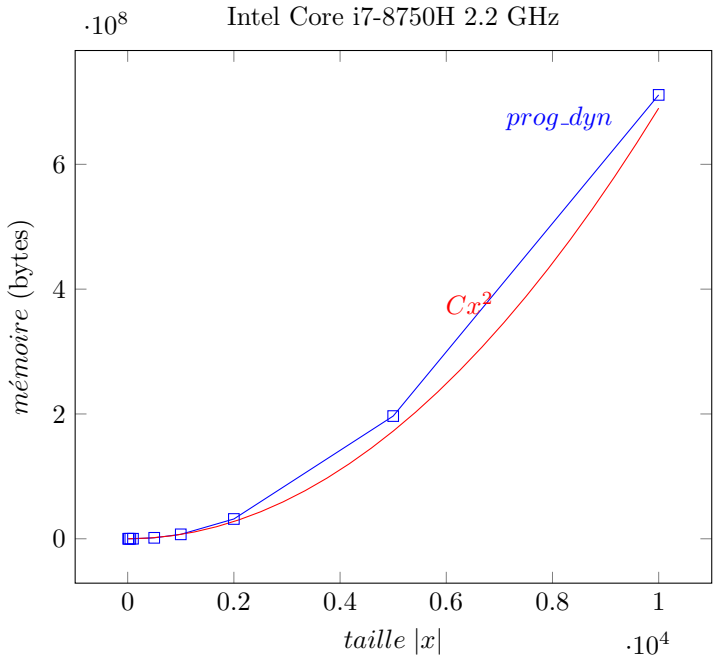




On a tracé la courbe de x^2 à une constante près (choisie très petite $\sim 10^{-7}$) à titre de comparaison. Le temps de calcul de **SOL_1** est assez proche de celui de **DIST_1** alors que sa complexité temporelle est plus faible. Cela vient du fait que pour **SOL_1** la complexité ne prend pas en compte le calcul du tableau T. On remarque une croissance en $O(x^2)$ pour les deux fonctions.
En ce qui concerne la taille maximale calculée, au moment où on arrive à des instances de la taille 100000 on obtient l'erreur suivante:

```
memory allocation of 712960 bytes failed
error: bench failed
```

On utilise la commande
GENOME_DATA=./tests/genome_instances_data/ valgrind –tool=massif ./target/debug/deps/mem_use-67d566998f108632 prog_dyn
pour mesurer l'usage mémoire de notre programme. Le main utilisé se trouve dans le fichier tests/mem_use.rs
La mémoire utilisée par la fonction **PROG_DYN** est de 711247899 bytes, soit 711MB pour l'instance Inst_0010000_7.adn.
Voici le graphe de l'estimation de la mémoire utilisée en fonction de la taille (sur lequel on a tracé la courbe de x^2):



Question 19 :

Expliquer pourquoi lors du remplissage de la ligne $i > 0$ du tableau T dans l'algorithme **DIST_1**, il suffirait d'avoir accès aux lignes $i-1$ et i du tableau (partiellement remplie pour cette dernière).

Dans **DIST_1** pour remplir la ligne i de T on utilise toujours une donnée de T qui est soit sur la même ligne i mais sur une colonne adjacente($T[i][j-1]$); soit sur la ligne précédente $i-1$ sur la même colonne ou sur une colonne adjacente($T[i-1][j-1]$ ou $T[i-1][j]$). On utilise donc pour remplir la ligne i seulement cette même ligne et la ligne précédente.

Question 20 :

En utilisant la remarque de la question précédente, donner le pseudo-code d'un algorithme itératif **DIST_2**, qui a la même spécification que **DIST_1**, mais qui a une complexité spatiale linéaire (en $\Theta(m)$).

Procedure 3 DIST_2

Input: x, y

$n \leftarrow |x| + 1$

$m \leftarrow |y| + 1$

T tableau 2D de taille $2 \times m$ // (en réalité un tableau des pointeurs vers deux autres tableaux sur le tas, ce qui permet d'utiliser swap)

$T[0][0] \leftarrow 0$

for j allant de 1 à m **do**

$T[0][j] \leftarrow T[0][j-1] + c_{ins}$

end for

for i allant de 1 à n **do**

$T[1][0] \leftarrow T[0][0] + c_{del}$

for j allant de 1 à m **do**

$$T[1][j] \leftarrow \min\begin{pmatrix} T[0][j-1] + c_{sub}(x_{i-1}, y_{j-1}), \\ T[0][j] + c_{del}, \\ T[1][j-1] + c_{ins} \end{pmatrix}$$

end for

swap(T[0], T[1]) // échange les pointeurs des lignes $T[0]$ et $T[1]$ en $O(1)$

end for

Output: $T[0][|y|]$

Ici on utilise un tableau de taille $2 \times m$ car seules deux lignes de T sont nécessaires à chaque étape de l'algorithme, et les anciennes lignes ne sont plus utilisées. Cela réduit la complexité spatiale en $\Theta(m)$ au lieu de $\Theta(n \times m)$ pour **DIST_1**.

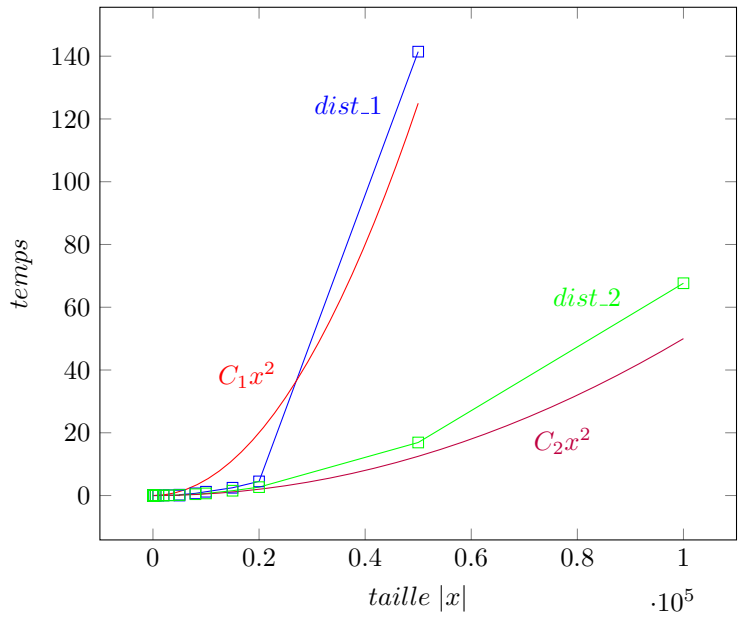
Question tache C :

- Coder la fonction **DIST_2** et la tester sur plusieurs instances.
- Tracer la courbe de consommation de temps CPU en fonction de la taille $|x|$ du premier mot du couple des instances fournies. Est-ce que la courbe obtenue correspond à la complexité théorique ? Vous pouvez vous limiter aux instances nécessitant moins de 10 minutes chacune.
- Comparer les résultats à ceux obtenus pour la fonction **DIST_1** précédente.
- Estimer la quantité de mémoire utilisée par **DIST_2** pour une instance de très grande taille.

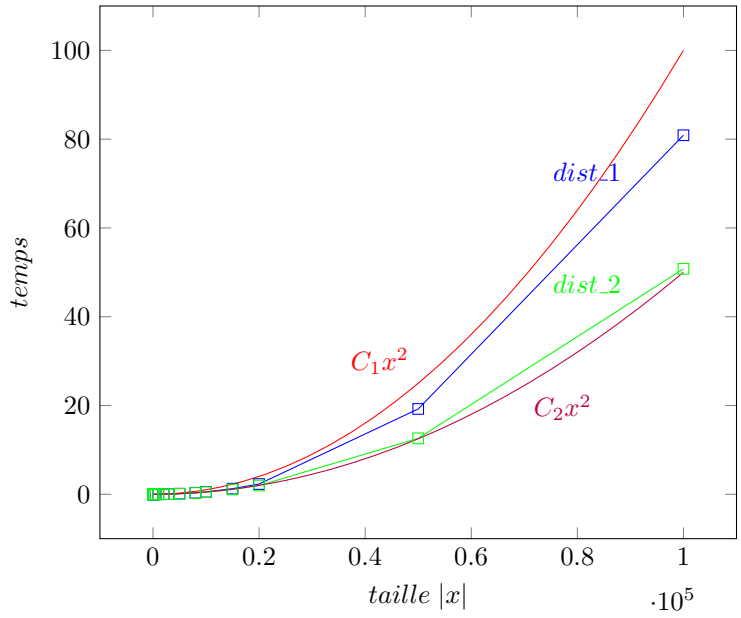
On utilise la commande pour générer les points:

```
$ cargo bench -- gnuplot dist_2
```

Intel Cote i7-8750H 2.2 GHz



Macbook M1

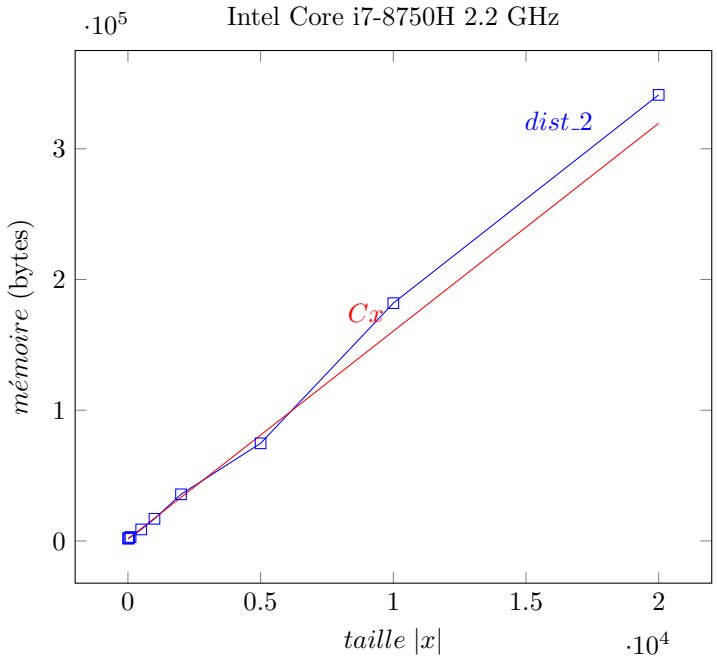


On utilise la commande :

GENOME_DATA=./tests/genome_instances_data/ valgrind –tool=massif ./target/debug/deps/mem_use-67d566998f108632 dist_2

pour mesurer l'usage mémoire de notre programme. Le main utilisé se trouve dans le fichier tests/mem_use.rs. La mémoire utilisée par la fonction **DIST_2** est de 324527 bytes, soit 324KB pour l'instance Inst_0020000_64.adn

Voici le graphe de l'estimation de la mémoire utilisée en fonction de la taille (sur lequel on a tracé la courbe de x à une constante près):



Question 21 :

Donner le pseudo code d'une fonction ***mot_gaps*** qui, étant donné un entier naturel k, renvoie le mot constitué de k gaps.

Procedure 4 mot_gaps

Input: k
  $mot \leftarrow \varepsilon$
  for i allant de 1 à k (inclu) **do**
    $mot \leftarrow mot \cdot -$
  end for
Output: mot

Question 22 :

Donner le pseudo code d'une fonction ***align_lettre_mot*** qui, étant donné x un mot de longueur 1 et y un mot non vide de longueur quelconque, renvoie un meilleur alignement de (x, y).

Idée : On cherche l'élément y_i de de y qui minimise la fonction $k \mapsto c_{sub}(y_k, x_0)$. Ensuite on compare notre résultat avec l'autre option, c'est-a-dire d'utiliser la suppression. Pour cela on compare $c_{sub}(y_k, x_0)$ obtenu a c_{del}. Si on trouve que $c_{sub}(y_k, x_0)$ est inférieur, alors on fait une substitution. En effet, une substitution va nous garantir un coût minimal dans ce cas. Cela consiste a $\overline{y} \leftarrow y$ et construire un $\overline{x}$ tel que $\forall k \neq i, \overline{x}_k = -$ et $\overline{x}_i = x_0$. Sinon on fait une suppression. Une suppression va consister a ajouter un gap à y pour obtenir $\overline{y}$ et mettre x en face de ce gap dans $\overline{x}$. On met des gaps dans $\overline{x}$ pour couvrir tous le reste.

Procedure 5 align_lettre_mot

Input: x, y ▷ $|x| = 1$ et $|y| \geq 1$

```
i ← 0
for t allant de 0 à |y| (exclu) do
    if c_sub(x_0, y_t) < c_sub(x_0, y_i) then
        i ← t
    end if
    if c_sub(x_0, y_t) = 0 then
        i ← t
        break
    end if
end for
if c_sub(x_0, y_i) < c_del then
    x̄ ← mot_gaps(|y|)
    x̄_i = x_0
    ȳ = y
else
    x̄ ← mot_gaps(|y| + 1)
    x̄_0 = x_0
    ȳ = − · y
end if
```

Output: $\overline{x}$, $\overline{y}$

Remarque : Cette autre version conserve la taille du mot nécessaire pour le bon fonctionnement de **SOL_2**:

Procedure 6 align_lettre_mot2

Input: x, y ▷ $|x| = 1$ et $|y| \geq 1$

```
i ← 0
for t allant de 0 à |y| (exclu) do
    if c_sub(x_0, y_t) < c_sub(x_0, y_i) then
        i ← t
    end if
    if c_sub(x_0, y_t) = 0 then
        i ← t
        break
    end if
end for
x̄ ← mot_gaps(|y|)
x̄_i = x_0
ȳ = y
```

Output: $\overline{x}$, $\overline{y}$

Question 23 :

Soit $x = BALLON$ et $y = ROND$. On pose $x^1 = BAL$, $x^2 = LON$, $y^1 = RO$, $y^2 = ND$.

1. Donner $(\overline{s}, \overline{t})$ un alignement optimal de (x^1, y^1).
2. Donner $(\overline{u}, \overline{v})$ un alignement optimal de (x^2, y^2).
3. Montrer que $(\overline{s} \cdot \overline{u}, \overline{t} \cdot \overline{v})$ n'est pas un alignement optimal de (x, y).

1. $(\overline{s}, \overline{t}) = (BAL,\ RO-)$ où $d = 13$.
2. $(\overline{u}, \overline{v}) = (LON-,\ --ND)$ où $d = 9$.
3. $(\overline{s} \cdot \overline{u}, \overline{t} \cdot \overline{v}) = (BALLON-,\ RO---ND)$ de coût $5+5+3+3+3+3 = 22$. Pourtant $(BALLON-,\ R---OND)$ est aussi un alignement de (x, y) mais son coût est $5+3+3+3+3 = 17 < 22$. Donc $(\overline{s} \cdot \overline{u}, \overline{t} \cdot \overline{v})$ n'est pas un alignement optimal.

Question 24 :

En supposant disposer de la fonction **coupure**, donner le pseudo-code de l'algorithme récursif de type diviser pour régner, nommé **SOL_2**, qui à partir d'un couple de mots (x, y) calcule un alignement minimal de (x, y).

Procedure 7 SOL_2(x, y)

if $|x| = 0$ **then**
  return (mot_gaps$(|y|), y)$
else if $|y| = 0$ **then**
  return $(x,$ mot_gaps$(|x|))$
else if $|x| = 1$ **then**
  return align_lettre_mot2(x, y)
else
  $i \leftarrow \left\lfloor \frac{|x|}{2} \right\rfloor$
  $j \leftarrow coupure(x, y)$
  $(x^1, y^1) \leftarrow SOL\_2(x_{[0..i[}, y_{[0..j[})$
  $(x^2, y^2) \leftarrow SOL\_2(x_{[i..|x|[}, y_{[j..|y|[})$
  return $(x^1 \cdot x^2, y^1 \cdot y^2)$
end if

Question 25 :

Donner le pseudo-code de la fonction **coupure**.

Procedure 8 coupure

Input: x, y

$i^* \leftarrow \left\lfloor \frac{|x|}{2} \right\rfloor$

T, I tableaux 2D de taille $2 \times m$ // (en réalité ce sont des tableaux des pointeurs vers deux autres tableaux sur le tas, ce qui permet d'utiliser swap)

$T[0][0] \leftarrow 0$

$I[0][0] \leftarrow 0$

for j allant de 1 à $|y|$ **do**

  $T[0][j] \leftarrow T[0][j-1] + c_{ins}$

  $I[0][j] = j$

end for

for i allant de 1 à $|x|$ **do**

  $T[1][0] \leftarrow T[0][0] + c_{del}$

  for j allant de 1 à $|y|$ **do**

    $\text{op1} \leftarrow T[0][j-1] + c_{sub}(x_{i-1}, y_{j-1})$

    $\text{op2} \leftarrow T[0][j] + c_{del}$

    $\text{op3} \leftarrow T[1][j-1] + c_{ins}$

    $T[1][j] \leftarrow \min(op1, op2, op3)$

    if $i > i^*$ **then**

      if $T[1][j] = \text{op1}$ **then**

        $I[1][j] = I[0][j-1]$

      else if $T[1][j] = \text{op2}$ **then**

        $I[1][j] = I[0][j]$

      else

        $I[1][j] = I[1][j-1]$

      end if

    end if

  end for

  swap(T[0], T[1]) // échange les pointeurs des lignes $T[0]$ et $T[1]$ en $O(1)$

  if $i > i^*$ **then**

    swap(I[0], I[1]) // échange les pointeurs des lignes $I[0]$ et $I[1]$ en $O(1)$

  end if

end for

Output: $I[0][|y|]$

Question 26 :

Quelle est la complexité spatiale de **coupure**?

La fonction utilise deux tableaux de taille $2 \times m$. Les autres variables sont de taille constante. Donc la complexité mémoire est de $\Theta(m)$

Question 27 :

Quelle est la complexité spatiale de **SOL_2**?

A chaque appel $SOL\_2(x, y)$ avec $|x| = n$ et $|y| = m$ la mémoire utilisée est de l'ordre de $\Theta(n+m)$. En effet la mémoire de la fonction coupure est en $\Theta(m)$. De plus $(x^1 \cdot x^2, y^1 \cdot y^2)$ est de taille $\Theta(n+m)$ dans le pire des cas d'après la question 2. La fonction récurrente s'écrit donc

$$T(n, m) = T(n/2, m * s_n) + T(n/2, m * (1 - s_n) + \Theta(n + m) \text{ avec } \forall n, s_n \in [0, 1]$$

En posant $k = n + m$, on peut l'approcher par $T(k) = 2T(k/2) + \Theta(k)$. En appliquant le théorème maître on trouve: $\Theta(k \log_2 k)$. Donc la complexité est de $\Theta((n+m) \log_2(n+m))$ (En réalité la hauteur de l'arbre des appels est en $\Theta(n)$ donc on pourrait montrer que la complexité est en $\Theta((n+m) \log_2 n)$)

Question 28 :

Quelle est la complexité temporelle de **coupure**?

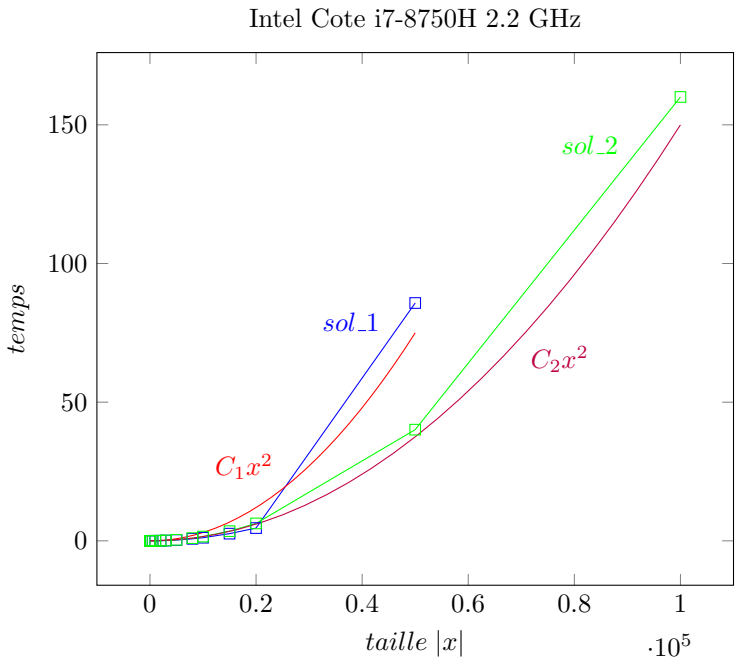
On effectue $n \times m$ itérations. Les autres boucles sont d'ordre moins important, enfin les opérations sont en temps constant. Ainsi la complexité est $\Theta(n \times m)$.

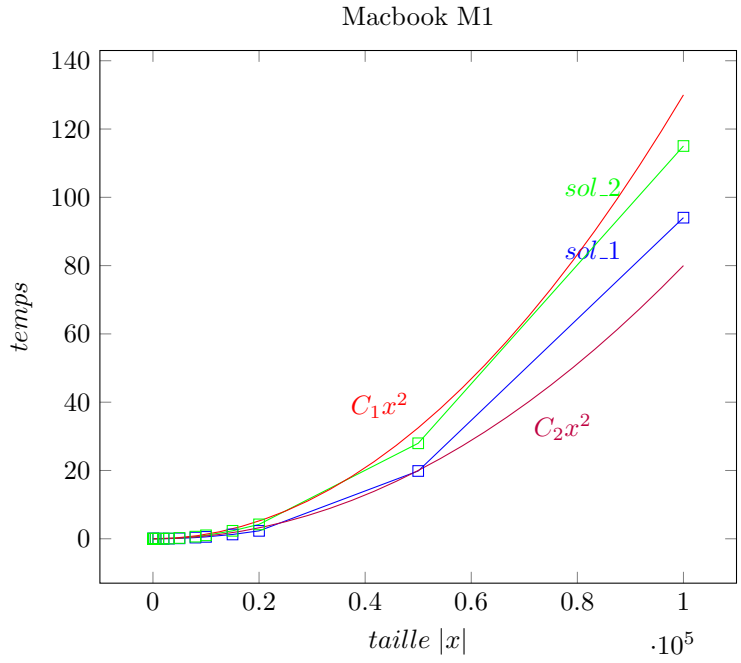
Question tache D :

- Coder les fonctions correspondant à cette méthode "diviser pour régner".
- Tracer la courbe de consommation de temps CPU en fonction de la taille $|x|$ du premier mot du couple des instances fournies. Vous pouvez vous limiter aux instances nécessitant moins de 10 minutes chacune.
- Estimer la consommation mémoire nécessaire au fonctionnement de cette méthode.

On utilise la commande:

```
$ cargo bench -- gnuplot sol_2
```



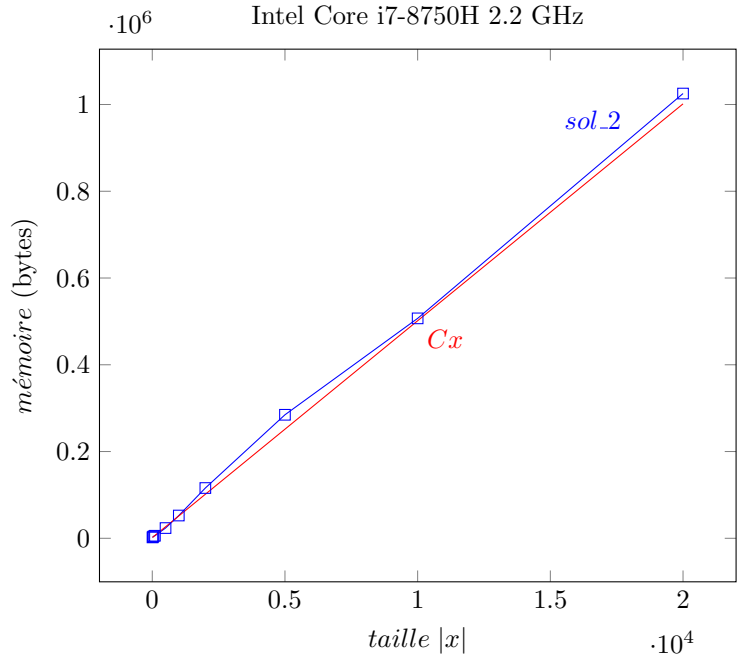


On utilise la commande

GENOME_DATA=./tests/genome_instances_data/ valgrind –tool=massif ./target/debug/deps/mem_use-67d566998f108632 sol_2

pour mesurer l'usage mémoire de notre programme. Le main utilisé se trouve dans le fichier tests/mem_use.rs. La mémoire utilisée par la fonction **SOL_2** est de 1025255 bytes, soit 1MB pour l'instance Inst_0020000_64.adn.

On a tracé la courbe de la mémoire utilisée en fonction de la taille et celle de la fonction x à titre de comparaison.



Question 29 :

A-t-on perdu en complexité temporelle en améliorant la complexité spatiale?
Comparez expérimentalement la complexité temporelle de **SOL_2** à celle de **SOL_1**. On ne demande pas de preuve théorique.

Non, en effet la fonction de complexité spatiale récursive de **SOL_2** s'écrit $T(n,m) = T(n/2, m \times s_n) + T(n/2, m * (1 - s_n) + \Theta(n \times m)$ avec $s_n \in [0,1]$ $\forall n$. En majorant par $k = \max(n,m)$ et en employant une technique similaire à celle de la question 27, on peut montrer que cela est très proche a $T(k) = 2T(k/2) + O(k^2)$. Le théorème maître donne alors: $\Theta(k^2)$ ce qui est dans le même ordre de grandeur que $\Theta(n \times m)$ selon l'ordre des paramètres.
La complexité reste la même, cependant on remarque sur le graphe que la constante C_2 devant la complexité de **SOL_2** est plus petite que celle de **SOL_1**. On pourrait penser que cela serait le contraire. En effet l'analyse théorique suggère une constante plus importante pour **SOL_2**. Cependant on ne doit pas oublier les allocations mémoire nécessaires pour la fonction **SOL_1**. Ces dernières ont une constante beaucoup plus importante.

5 Une extension : l'alignement local de séquences (Bonus)

Question 30 :

En générant quelques instances, constater expérimentalement que le coût d'un alignement global de (x, y) quand $|y| << |x|$ (relativement à $|\Sigma|$) est $(|x||y|)c_{\text{del}}$.

On a vérifie expérimentalement ce fait avec une instance qu'on a construit *Instance_long_short.adn*. Le test se trouve dans math.rs dans bonus_q30. Cette fonction confirme que le coût de l'alignement correspond bien a la formule indiquée.

Question 31 :

Cela semble-t-il une bonne idée ? Vous pouvez par exemple vous demander quel serait le coût d'un plus long alignement que vous avez proposé en exemple à la question 2.

En suivant cette idée, l'alignement de la question 2 serait minimal de coût 0. Cela impliquerait que tout alignement retourné par une des fonctions d'alignement serait toujours l'alignement de longueur maximale. Ce qui n'est pas très intéressant.

Question 32 :

En s'inspirant de la partie 3, comment pourrait-on résoudre les problèmes BEST_SCORE et ALI_LOC en complexité spatiale $\Theta(n \times m)$? Avec quelles complexités temporelles ? Pourrait-on alors réduire la complexité spatiale par une méthode diviser pour régner sans dégrader la complexité temporelle ? On attend une réponse relativement courte, dites surtout ce qui peut s'adapter et comment, et au contraire ce qui ne peut pas s'adapter et pourquoi.

On ne modifie que les coûts différents, il est donc possible d'adapter les fonctions se basant sur la programmation dynamique. On pourrait alors très bien adapter **dist_2**, **dist_1**, **prog_dyn** et **sol_1** pour ces nouvelles définitions. Cependant la fonction **coupure** ne fonctionnerait pas correctement pour des appels sur les sous-séquences car elle considérerait toujours que les coûts des gaps au début et à la fin nuls, ce qui, par exemple, n'est pas vrai pour une sous-séquence qui commence au milieu de sa séquence mère (en effet pour cette séquence les coûts au début ne sont pas forcément nuls). Pour palier à cela on pourrait passer deux paramètres supplémentaires. Plus précisément des *flags*

qui mettraient les coûts en début (resp. en fin) a zéro. Cela permettrait de faire les appels récursifs en conservant les bons coûts pour les sous-séquences.