Sujet : Pré-traitement algorithmique et applications

Il est rappelé que le jury attend un exposé de 35 minutes, pédagogique et structuré, fondé sur ce sujet. Ce texte se conclut par une liste de pistes de réflexion pour guider la conception de l'exposé, mais toute initiative personnelle pertinente est appréciée. L'exposé doit contenir une ou plusieurs illustrations informatiques ainsi qu'une discussion autour d'une dimension éthique, sociétale, environnementale, économique ou juridique. Une piste particulière est proposée à cet effet.

1 Introduction

Lorsque l'on souhaite effectuer des requêtes de nombreuses fois sur les mêmes données, il est souvent avantageux de passer du temps à préparer les données, même si cela induit un surcoût initial en temps ou en espace, pour gagner globalement en efficacité ensuite.

Un des exemples les plus simples est le problème RECHERCHE(A,x) qui consiste à décider si un élément x est dans un tableau A de taille n, contenant des éléments deux à deux comparables. Si on ne fait qu'un appel à RECHERCHE, une recherche séquentielle résout le problème en $\mathcal{O}(n)$ comparaisons. Mais si on fait t recherches et que t est grand, on a intérêt à commencer par trier A, pour effectuer ensuite chaque recherche en temps $\mathcal{O}(\log n)$. La complexité de cette solution est en $\mathcal{O}(n \log n + t \log n)$, contre $\mathcal{O}(tn)$ si on effectue chaque recherche de façon séquentielle. Ainsi, le tri du tableau est vu comme un *pré-traitement* qui permet d'optimiser les requêtes de recherche effectuées ultérieurement.

Dans la suite, nous commençons par étudier un problème purement algorithmique, dans ce paradigme. Cela consiste à trouver la plus petite valeur entre deux indices dans un tableau. Ce problème nous servira ensuite de brique de base pour obtenir des solutions efficaces à plusieurs applications.

2 Calcul des minimas dans les plages d'un tableau

On utilise également les notations PYTHON pour décrire les sous-tableaux : si T est un tableau de taille n et $0 \leq i \leq j < n$, alors $T[\,:i]$ est le sous-tableau constitué des i premières valeurs de

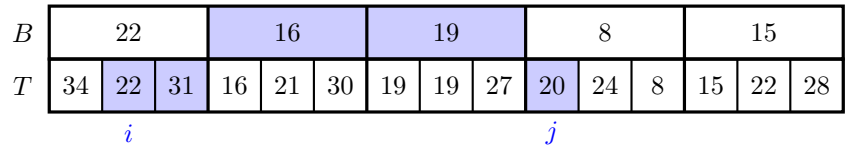


FIGURE 1 – RMQ avec découpage en blocs. Une fois le minimum de chaque bloc calculé et stocké dans B, on peut répondre plus rapidement : il suffit de prendre le minimum entre les valeurs à partir de i et jusqu'à j dans leurs blocs respectifs (donc 22 et 20), et du minimum des valeurs de B entre leurs deux blocs (ici, 16), comme indiqué ci-dessus avec $\ell = 4$ blocs de taille $b = 3$.

T, $T[i:]$ celui constitué des $n-i$ dernières valeurs de T et $T[i:j]$ des valeurs entre les indices i et $j-1$.

On souhaite effectuer un pré-traitement sur un tableau de nombres T, de taille n, afin de pouvoir renvoyer $\mathrm{RMQ}(T,i,j) = \min_{i \le k \le j} T[k]$ en temps constant, pour $0 \le i \le j < n$.

Une première idée naïve pour résoudre ce problème est de calculer et stocker le résultat pour tous les couples (i,j). On peut alors répondre en temps $\mathcal{O}(1)$ aux requêtes $\mathrm{RMQ}(T,i,j)$. Cependant, le précalcul est trop coûteux à la fois en temps et en espace pour que cette solution soit envisageable dans les applications qui utilisent cette structure. On souhaite que le précalcul se fasse en temps linéaire et en utilisant un espace additionnel linéaire.

2.1 Découpage en blocs

Pour diminuer le temps de pré-calcul, on peut découper en blocs de b cases consécutives. On obtient alors $\ell \approx n/b$ tels blocs. On peut calculer et stocker le minimum de chaque bloc dans un tableau B en temps global $\mathcal{O}(n)$ et en espace $\mathcal{O}(\ell)$.

Une fois B construit, on répond aux requêtes en calculant le minimum des valeurs de T entre i et la fin de son bloc et entre j et le début de son bloc, ainsi que le minimum dans B pour les blocs situés entre ceux de i et j, comme représenté sur la figure 1.

En prenant b de l'ordre de $\sqrt{n}$, on peut ainsi faire un pré-traitement linéaire pour calculer B, obtenir un espace additionnel en $\mathcal{O}(\sqrt{n})$ et effectuer les requêtes $\mathrm{RMQ}(T,i,j)$ en temps $\mathcal{O}(\sqrt{n})$. De façon générale, l'espace additionnel est en $\mathcal{O}(\ell)$ et le temps des requêtes en $\mathcal{O}(b+\ell)$.

2.2 Ajout d'une table à chaque bloc

Pour l'instant, on a stocké dans B le minimum de chaque bloc. On va maintenant calculer et stocker, pour chaque indice t de B et pour chaque k entre 0 et $\lfloor \log_2 \ell \rfloor$, les quantités (avec $B[s] = +\infty$ si on sort du tableau) :

$$B^+[t][k] = \min_{0 \le s < 2^k} B[t+s] \text{ et } B^-[t][k] = \min_{0 \le s < 2^k} B[t-s].$$

On regarde ainsi des plages d'indices partant ou arrivant en t, dont les longueurs sont des puissances de 2. En s'y prenant bien, on peut calculer chaque valeur de B^+ et de B^- en temps $\mathcal{O}(1)$. Ce qui fait donc une complexité totale en temps et en espace en $\mathcal{O}(\ell \log \ell)$.

Une fois qu'on a calculé B^+ et B^-, on peut répondre à $\mathrm{RMQ}(B,s,t)$ en temps $\mathcal{O}(1)$: si k est

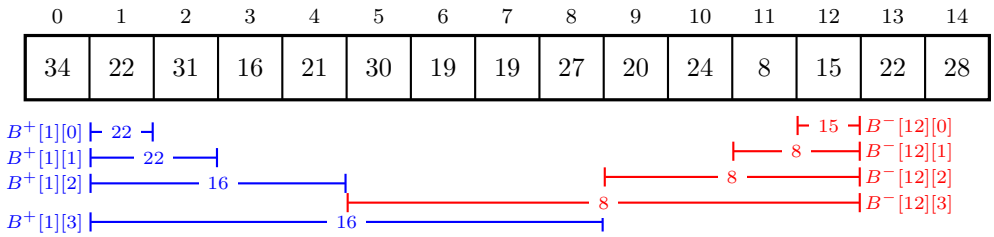


FIGURE 2 – La ligne $B^+[1]$ contient les minima des plages de longueurs 1, 2, 4, ... qui commencent en 1, et la ligne $B^-[12]$ contient les minima des plages de longueurs 1, 2, 4, ... qui terminent en 12. On a donc $\text{RMQ}(B, 1, 12) = \min(B^+[1][3], B^-[12][3]) = 8$.

le plus petit entier positif tel que $2^k \geq \frac{t-s-1}{2}$ alors les indices des calculs de $B^+[s][k]$ et $B^-[t][k]$ recouvrent exactement $\{s, s+1, \ldots, t\}$, et donc $\text{RMQ}(B, s, t) = \min(B^+[s][k], B^-[t][k])$ (voir la figure 2 pour un exemple). On suppose qu'on a accès à la bonne valeur de k en temps constant, par exemple avec un autre pré-calcul.

2.3 Combinaison des deux idées

On combine les deux idées de la façon suivante :

1. On reprend le découpage en blocs de la partie 2.1, en fixant une taille de blocs b en $\Theta(\log n)$. Le tableau B contient donc $\ell = \Theta(\frac{n}{\log n})$ blocs.
2. On calcule les tables B^+ et B^- de la partie 2.2 pour le tableau B. Cela se fait en temps et espace $\mathcal{O}(n)$ pour le choix de b en $\Theta(\log n)$.
3. Pour répondre aux requêtes $\text{RMQ}(T, i, j)$, on calcule le minimum entre i et la fin de son bloc en temps $\mathcal{O}(\log n)$, entre j et le début de son bloc en temps $\mathcal{O}(\log n)$, et la partie intermédiaire dans B en temps $\mathcal{O}(1)$ comme expliqué à la fin de la partie 2.2.

On a ainsi un pré-traitement effectué en temps $\mathcal{O}(n)$ avec un espace supplémentaire en $\mathcal{O}(n)$, pour des requêtes $\text{RMQ}(T, i, j)$ qui se font en temps $\mathcal{O}(\log n)$.

2.4 Arbres Cartésiens

La majeure partie du temps de calcul de notre solution actuelle pour $\text{RMQ}(T, i, j)$ réside donc dans le calcul du minimum de T entre i et la fin de son bloc, et entre j et le début de son bloc. Une idée pourrait être d'utiliser la méthode naïve en précalculant et stockant toutes les possibilités sur chaque bloc, pour réduire le temps de requête à $O(1)$. Un tel précalcul coûterait un temps et un espace additionnel en $\mathcal{O}(n \log n)$, toujours pour $b = \Theta(\log n)$. On va rendre cette idée efficace en utilisant les arbres Cartésiens.

Deux tableaux d'entiers S_1 et S_2 de taille b sont dit *équivalents* lorsque pour tous indices $i \leq j$ de $\{0, \ldots, b-1\}$ les minima de S_1 et S_2 entre les indices i et j sont situés à la même position (s'il y a des égalités, on choisit la position la plus à gauche comme minimum). On le note $S_1 \equiv S_2$. On peut vérifier par exemple que $[4, 1, 2, 3] \equiv [2, 1, 3, 4]$.

On calcule l'équivalence des blocs en passant par les *arbres Cartésiens*. L'arbre cartésien $\mathbb{A}(T)$ d'un tableau T de nombres est défini inductivement par :

- si $|T| = 0$, alors $\mathbb{A}(T) = \texttt{nil}$, l'arbre vide ;
- sinon, $\mathbb{A}(T) = \overset{T[i]}{\underset{\mathcal{G}\ \ \mathcal{D}}{/\backslash}}$, avec $\mathcal{G} = \mathbb{A}(T[\,:i])$ et $\mathcal{D} = \mathbb{A}(T[i+1:\,])$ et où i est l'indice du minimum de T (en cas d'égalité, on prend le plus petit i).

On appelle la *forme* d'un arbre binaire, l'arbre obtenu en effaçant les étiquettes de ses nœuds.

Lemme 1 *Deux tableaux d'entiers sont équivalents si et seulement si leurs arbres Cartésiens ont même forme.*

ArbreCartésien(S)

```
1  P ← pile_vide()
2  pour x ∈ S dans l'ordre faire
3  |  N ← x /\ nil nil
4  |  N' ← nil
5  |  tant que P ≠ ∅ et P.top.racine > x
   |   faire
6  |  |  N' ← depiler(P)
7  |  N.fils_gauche ← N'
8  |  si P ≠ ∅ alors
9  |  |  P.top.fils_droit ← N
10 |  P.empiler(N)
11 renvoyer P.top
```

On en déduit un procédé algorithmique pour tester si deux blocs S_1 et S_2 sont équivalents : on calcule $\mathbb{A}(S_1)$ et $\mathbb{A}(S_2)$, puis on teste s'ils ont même forme. On peut calculer l'arbre Cartésien en temps linéaire avec l'algorithme ArbreCartésien ci-contre.

On utilise une pile qui est initialisée avec `pile_vide()` et dont on accède à l'élément du haut (le plus récemment ajouté) avec `.top`. Plus précisément, P.`top.racine` renvoie la valeur de la racine de l'arbre en haut de la pile P, et P.`top.fils_droit` est son fils droit.

On peut ensuite encoder l'arbre $\mathbb{A}(S)$ avec un mot binaire de longueur $2b$, où $b = |S|$, en utilisant l'ordre d'un parcours préfixe et, quand on découvre un nouveau nœud qui n'est pas `nil`, on écrit 00 si ses deux fils sont `nil`, 10 si son fils droit est `nil` mais pas son fils gauche, 01 si son fils gauche est `nil` mais pas son fils droit, et 11 si ses deux fils sont différents de `nil`. On notera $\varphi(\mathbb{A}(S))$ cet encodage de l'arbre. Par exemple, $\varphi(\mathbb{A}([4,1,2,3])) = \texttt{11000100}$. On peut montrer que deux arbres ont la même forme si et seulement si ils ont le même encodage.

2.5 Finalisation de la construction

On peut maintenant finaliser la construction : on fixe $b = \lceil \frac{1}{2} \log_4 n \rceil$. On repart de la construction présentées dans la partie 2.3 avec cette valeur de b. Ensuite, pour chaque bloc β de longueur b de T :

— On lui associe l'encodage son arbre Cartésien $\varphi(\mathbb{A}(\beta))$.

— Si c'est la première fois qu'on voit cet encodage, on calcule pour chaque couple $0 \leq i \leq j < b$ l'indice du minimum de β entre les indices i et j, de façon naïve. On associe le résultat à $\varphi(\mathbb{A}(\beta))$.

Ainsi, en passant par l'encodage de son arbre Cartésien qui est pré-calculé, on a accès à la positions de tous les minima à l'intérieur de chaque bloc de T.

L'amélioration de complexité vient du fait qu'il y a moins de $2^{2b} = \mathcal{O}(\sqrt{n})$ encodages différents. Les pré-calculs des minima des différents encodages prennent donc un temps total $o(n)$. Cela finalise la construction, puisque l'on peut maintenant trouver le minimum entre i et la fin de son bloc et entre j et le début de son bloc en temps constant. On a donc le résultat ci-dessous.

Théorème 1 *Soit T un tableau de nombres, de taille n. En temps $\mathcal{O}(n)$, on peut effectuer un pré-traitement qui utilise un espace additionnel en $\mathcal{O}(n)$, qui permet de répondre à chaque requête* $\mathrm{RMQ}(T, i, j)$ *en temps constant.*

3 Application en algorithmique du texte

Le problème classique de recherche de motif dans un texte consiste à décider si un mot X de longueur m (le *motif*) est facteur d'un mot T de longueur n (le *texte*), c'est-à-dire s'il existe des mots Y et Z tels que $T = YXZ$. On peut résoudre ce problème en temps $\mathcal{O}(n + m)$ avec, par exemple, l'algorithme de Knuth-Morris-Pratt (qui effectue un pré-traitement linéaire sur le motif X).

Dans notre cadre d'étude le problème est de savoir s'il est possible d'effectuer un pré-traitement sur T pour accélérer la recherche, quand on va l'effectuer pour de nombreux motifs. On dispose par exemple d'une longue séquence d'ARN de référence, que des chercheurs étudieront en utilisant plusieurs motifs, que l'on ne connaît pas à l'avance.

Dans toute la suite, on considère que l'alphabet est fixé, et on ne fait donc pas intervenir sa taille dans les complexités.

3.1 Recherche dans un tableau trié de chaînes de caractères

On commence par un problème plus simple : on suppose que A est un tableau contenant n mots, qui sont ordonnés selon l'ordre lexicographique $\leq_{\text{lex}}$. On note $\|A\|$ la somme des longueurs des mots de A. On veut un algorithme efficace pour tester si un mot X de longueur m est dans A.

Comme le tableau est trié, il est naturel de faire une dichotomie pour résoudre le problème. Mais ici, ce n'est pas optimal, on peut faire mieux en exploitant les propriétés de l'ordre lexicographique.

Pour deux mots u, v, on note $\mathrm{lcp}(u, v)$ le plus long préfixe commun à u et v. Il est possible d'améliorer la dichotomie en utilisant l'idée suivante : $\mathrm{lcp}(A[d], A[f])$ est préfixe de tous les mots de A entre les deux indices d et f. Donc si on suppose que grâce à un pré-calcul on a accès en temps constant à $\mathrm{lcp}(A[d], A[f])$, où d et f sont deux bornes de la dichotomie, alors quand on compare X avec $A[\lfloor (d + f)/2 \rfloor]$ pour l'ordre lexicographique, ce n'est pas la peine de regarder les $|\mathrm{lcp}(A[d], A[f])|$ premiers caractères.

En utilisant cette idée, on peut construire un algorithme qui a une bien meilleure complexité que la dichotomie habituelle, comme résumé dans le théorème suivant.

Théorème 2 *Si les requêtes* $\mathrm{LCP}(A, i, j)$ *se font en temps $\mathcal{O}(1)$, alors il existe une variante de la dichotomie qui décide si un mot X est dans A en temps $\mathcal{O}(m + \log n)$, avec $n = |A|$ et $m = |X|$.*

Il nous reste à réaliser le pré-traitement pour les requêtes $\mathrm{LCP}(A, i, j)$. On va tout d'abord construire un tableau `lcp` de longueur n, que l'on construit pour que `lcp[i]` $= \mathrm{LCP}(A, i, i+1)$ pour $i \leq n - 2$ et `lcp[n-1] = 0`. On calcule donc le plus long préfixe commun de chaque mot avec le suivant. Cette étape peut être réalisée en temps $\mathcal{O}(\|A\|)$ en comparant $A[i]$ et $A[i + 1]$ caractère par caractère. L'utilité de `lcp` réside dans la propriété suivante :

Lemme 2 *Pour tout* $0 \leq i < j < n$, *on a* $\mathrm{LCP}(A, i, j) = \min_{i \leq k < j} \texttt{lcp}[\texttt{i}] = \mathrm{RMQ}(\texttt{lcp}, i, j-1)$.

Il suffit donc d'utiliser, en pré-traitement, les constructions de la partie 2 sur le tableau `lcp` pour être dans les hypothèses du théorème 2 et faire les recherches en temps $\mathcal{O}(m + \log n)$.

Remarque. On aurait pu résoudre le problème plus efficacement en construisant un automate déterministe qui reconnaît l'ensemble des mots de A, ou bien un arbre préfixiel avec les mots de A. Mais cette solution ne peut pas être généralisée à la partie suivante, contrairement à l'algorithme du théorème 2.

3.2 Indexation d'un texte

Retournons au problème initial de cette partie, qui consiste à effectuer un pré-traitement sur un texte T, afin d'optimiser des recherches de motifs X. On remarque d'abord que X est un facteur si et seulement si X est le préfixe d'un suffixe de T. Ensuite, il est immédiat d'adapter l'algorithme du théorème 2 pour décider si X est préfixe d'un mot du tableau A. Donc si on peut trier les suffixes de T, et calculer le tableau `lcp` associé, le problème est résolu.

Si on utilise un tri classique en $\mathcal{O}(n \log n)$ comparaisons pour les suffixes, la complexité est catastrophique dans le pire cas. Trier les suffixes d'un mot est un problème qui a été beaucoup étudié en algorithmique du texte, et dont les solutions sont trop compliquées pour être exposées dans ce document. On admettra donc le résultat suivant :

Théorème 3 *Il existe un algorithme de complexité* $\mathcal{O}(n)$ *qui calcule la permutation qui trie les indices des débuts des suffixes de* T *selon l'ordre lexicographique des suffixes correspondants et qui calcule également le tableau* `lcp` *associé.*

Au-delà de la recherche de motif, le théorème 3 a de nombreuses applications. Par exemple, pour rechercher des similarités entre deux séquences (utile en bio-informatique ou pour la détection de plagiat). Calculer le tableau `lcp` associé au tri des suffixes du mots $T_1 \# T_2$, où $\#$ est un symbole séparateur, peut être utilisé pour chercher des similitudes entre les textes T_1 et T_2.

4 Plus proche ancêtre commun

On peut se servir des constructions de la partie 2 en algorithmique des arbres. Le problème est le suivant : on a un arbre $\mathcal{A}$ et deux nœuds x et y de $\mathcal{A}$. On note $\mathrm{LCA}(\mathcal{A}, x, y)$ l'ancêtre commun à x et y qui est le plus proche de x et de y (et donc le plus loin de la racine). Dans l'exemple de la figure 3, on a $\mathrm{LCA}(\mathcal{A}, I, H) = F$.

Une simple transformation permet de se ramener à un problème de type RMQ. On va effectuer un parcours en profondeur pour remplir un tableau `dist`, de gauche à droite. Quand on découvre un nœud, on note son étiquette et sa distance à la racine dans `dist` ; on recopie également ces deux informations à chaque fois qu'on a terminé le traitement d'un de ces fils. Ainsi pour l'exemple de la figure 3, le nœud F est à distance 1 de la racine et possède 3 fils, il va donc contribuer pour 4 occurrences de $(F, 1)$ dans `dist`. On associe également à chaque nœud N l'indice dans `dist` de sa première contribution, qu'on note `indice[`N`]`.

Grâce à cette construction, on peut donc répondre aux requêtes $\mathrm{LCA}(\mathcal{A}, x, y)$ en temps constant après un traitement linéaire en le nombre de nœuds : il suffit d'identifier le sommet

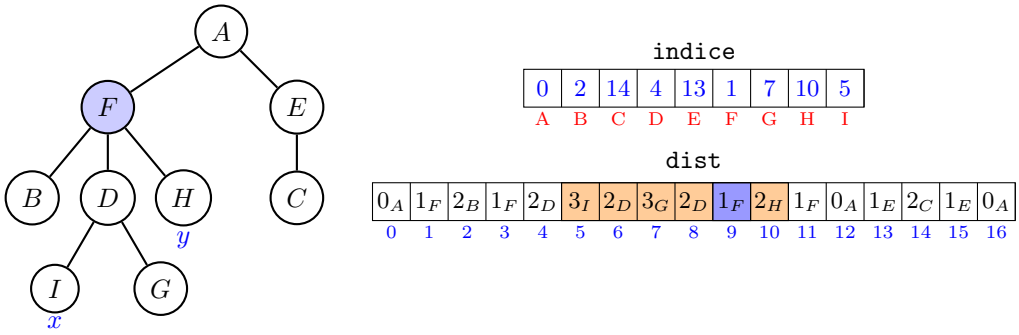
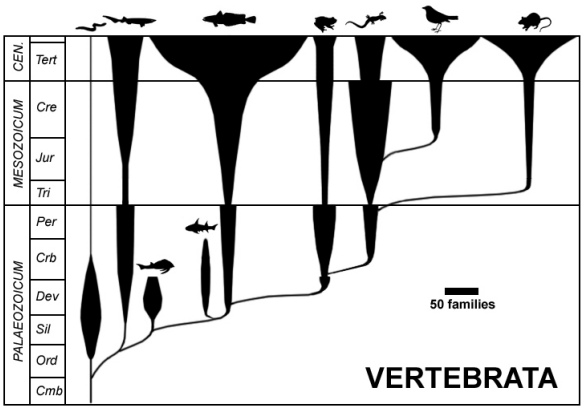


FIGURE 3 – La transformation en un problème RMQ. Le parcours en profondeur produit le tableau `dist`, dans lequel on a noté en indice le nom du nœud ayant contribué. Pour connaître le plus proche ancêtre commun à H et I, le tableau `indice` nous dit qu'il faut regarder `dist` entre les indices 5 et 10 : le minimum sur cette plage est réalisé par le nœud F.

de `dist` de valeur minimale entre les indices `indice[`x`]` et `indice[`y`]`. Parmi les applications de LCA il y a :

- Le calcul des plus courts chemins pour les routages dans des réseaux de capteurs, quand ils sont organisés en arbre. On utilise le fait que le plus court chemin dans un arbre entre deux nœuds x et y consiste à passer par leur plus proche ancêtre commun.
- 

L'étude informatique des arbres phylogénétiques, qui décrit les relations de parenté entre des groupes d'êtres vivants. Le plus proche ancêtre commun à deux espèces permet d'identifier où elles ont commencé à diverger.

  À gauche, une représentation des familles de vertébrés. La largeur est proportionnelle au nombre de familles regroupées dans la catégorie. Certains arbres phylogénétiques sont construits à partir de données telles que des séquences d'ADN ou de protéines et peuvent être de très grandes tailles.

Trouver rapidement le plus proche ancêtre commun à deux nœuds est également utile pour les logiciels de gestion de versions comme `git`. Il faut cependant développer une autre algorithmique, puisque les branchements et des fusions induisent naturellement une structure de graphe orienté acyclique, et non d'arbre.

Pistes de réflexion pour l'exposé

1. Implanter l'algorithme pour RMQ qui utilise les constructions de la partie 2.1. Est-ce que le choix de $b = \sqrt{n}$ est pertinent ? Expérimentalement, faut-il beaucoup de requêtes pour que ce soit plus intéressant que de calculer le minimum à chaque fois ?
2. Expliquer comment on peut calculer le tableau B^+ de la partie 2.2 en temps $\mathcal{O}(\ell \log \ell)$, où ℓ est la taille de B.
3. Implanter l'algorithme ArbreCartesien. Discutez de sa correction et de sa complexité.
4. Expliquer pourquoi deux arbres Cartésiens différents ont des encodages différents par φ, et pourquoi il y a moins de 2^{2b} formes d'arbres Cartésiens différentes.
5. Si A est un tableau de mots qui n'est pas ordonné, expliquer comment on peut le trier en temps $\mathcal{O}(\|A\|)$.
6. Discuter de solutions pour la détection de similarités dans deux textes, et de l'importance d'avoir des algorithmes pertinents et efficaces pour le faire.
7. Quelle est la complexité pire cas d'utiliser un tri en $\mathcal{O}(n \log n)$ comparaisons pour trier les suffixes d'un mot ? Est-ce que si ce mot est un texte en français, on peut s'attendre à un temps d'exécution beaucoup plus rapide que le pire cas ?
8. Implanter la transformation du problème LCA en problème RMQ. Justifier qu'elle est de complexité linéaire.
9. Expliquer pourquoi les valeurs de deux cases consécutives de `dist` ne diffèrent que de 1. Comment peut-on se servir de cette propriété pour simplifier la partie 2.4 si on sait que T possède cette propriété ?
10. Le projet "Icelandic Human Genome Project" a pour objectif de séquencer le génome de la population islandaise afin de mieux comprendre les maladies génétiques et de développer des traitements personnalisés. Discuter de la pertinence et des risques de stocker de nombreuses informations génétiques pour des traitements informatiques.