Sélection de sujets posés lors de la session 2025

Exercices de type A

Exercice 1 Course d'escargots (type A) On s'intéresse à l'organisation d'une course d'escargots. Dans ce milieu les paris peuvent monter haut et les enjeux importants poussent parfois au dopage des escargots participant à ces courses. Vous êtes chargé de proposer une méthode utilisant de l'apprentissage automatique pour mieux cibler les tests sur les escargots les plus susceptibles d'avoir été dopés.

Les seules informations disponibles concernant les escargots en amont de la course sont leur masse et la longueur de leur antennes.

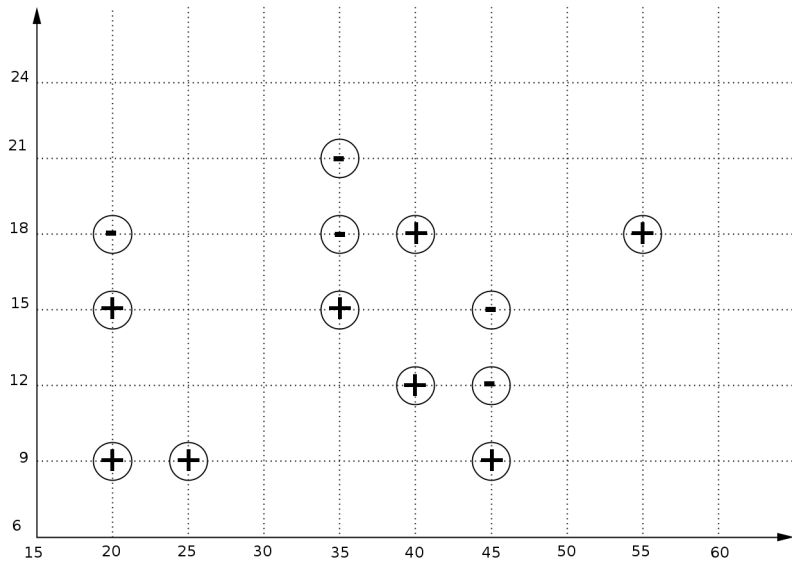
Pour vous aider dans votre tâche, l'organisation en charge de la course met à votre disposition les données passées concernant les tests réalisés, contenant les informations suivantes :

- Numéro du test
- Masse de l'escargot (g)
- Longueur des antennes (mm)
- Résultat du test

Sur la base de ces informations, il vous est demandé de proposer une approche permettant, en amont d'une course, de prédire pour chaque escargot s'il est ou non dopé.

1. De quel type de problème d'apprentissage s'agit-il ?
2. L'algorithme de classification hiérarchique ascendante serait-il pertinent pour résoudre ce problème ? Justifier brièvement votre réponse.

La figure (A) présente de manière graphique la liste des tests passés fournie par l'organisation de la course. La figure (B) est une liste d'escargots qui participeront à la prochaine course.



A : La liste des tests passés et leur résultat. Les tests positifs sont marqué d'un + et ceux négatifs sont marqué d'un -. En abscisse la masse et en ordonnée la longueur des antennes.

n° candidat	Masse	Longueur
0	20	12
1	50	12
2	40	15
3	30	21

B : Les escargots participant à la prochaine course.

On souhaite réaliser les prédictions à l'aide de l'algorithme des k plus proches voisins.

3. Rappelez le principe de cet algorithme et détaillez la manière dont on pourrait l'appliquer dans le cas présent.

4. On fixe $k = 3$ et on utilise la distance euclidienne. Quels sont les escargots pour lesquels ce modèle prédit un dopage possible ?

5. Quelle est la complexité temporelle d'une prédiction de cet algorithme dans le cas général (le nombre de points n et la valeur de k ne sont pas supposés fixés) ?

6. Est-ce possible d'espérer une complexité meilleure via l'usage d'une implémentation adaptée ?

Par conscience professionnelle, vous vous demandez si le modèle utilisé est efficace et en particulier si vous avez correctement choisi la valeur de k. Pour répondre à cette question, vous tentez de prédire le résultat du test des 10 premiers escargots d'une seconde liste, plus grande et fournie par l'organisation de la course, pour différentes valeurs de k.

Pour $k = 3$ vous obtenez les prédictions suivantes :

numéro du test	prédiction	résultat réel
0	négatif	négatif
1	négatif	positif
2	négatif	négatif
3	positif	négatif
4	positif	positif
5	négatif	négatif
6	positif	négatif
7	négatif	négatif
8	positif	négatif
9	positif	positif

7. Dressez la matrice de confusion de ces prédictions. Commentez le résultat obtenu.

À l'aide de nombreux calculs, vous obtenez le tableau suivant donnant le taux de prédictions correctes de votre modèle pour différentes valeur de k sur les 10 premiers tests de la seconde liste.

k	1	2	3	4	5	6	7
taux de prédictions correctes	30%	40%	70%	40%	30%	10%	5%

Vous choisissez donc de conserver $k = 3$.

8. Est-ce raisonnable d'affirmer que votre modèle a un taux de prédictions correctes de 70% ?

Proposition de corrigé

1. Il s'agit d'un problème d'apprentissage supervisé, et plus précisément de classification binaire pour prédire la valeur du test de dopage.
2. Non, puisque la classification hiérarchique ascendante est un algorithme d'apprentissage non supervisé.
3. Pour prédire la classe d'un point, on regarde la classe majoritaire parmi ses k plus proches voisins. Ici, on tente de prédire le résultat du test (la classe) à partir de la proximité vis à vis des caractéristiques connues en amont : masse et longueur des antennes. Sans précisions supplémentaire on pourra choisir la distance euclidienne sur le couple des entrées.
4. Il faut appliquer l'algorithme des k plus proches voisins. Un dessin permet de donner des réponses immédiates sans avoir besoin de réaliser des calculs. Les résultats prédits sont, dans l'ordre :

numéro du candidat escargot	Masse (en g)	Longueur des antennes (en mm)	prédiction de dopage
0	20	12	positive
1	50	12	négative
2	40	15	positive
3	30	21	négative

5. On doit trouver les k plus proches voisins, il y a plusieurs manières de procéder, mais la plus simple est de calculer toutes les distances, puis de trier la liste des distances obtenues et de prendre les k premières. La complexité du calcul des distances dans ce cas est en $O(n)$ et $O(dn)$ dans le cas général d'une distance euclidienne de dimension d. Le tri est en $O(n\log(n))$ via un algorithme adapté. Une prédiction est donc en $O(n\log(n))$.
6. On pourrait utiliser des arbres d-dimensionnels pour élaguer une partie de la recherche des voisins. Cela ne change rien au pire cas, mais peut améliorer la complexité dans certains cas. Si une file de priorité est proposée pour passer de $n\log(n)$ à $n\log(k)$ est proposée, on demande une autre structure plus spécifique au problème et on ne donne les points que si les arbres d-dimensionnels sont connus.
7. pour la matrice.

   On obtient la matrice suivante :

	prédiction test positif	prédiction test négatif
test positif	2	1
test négatif	3	4

   pour le commentaire.

   On peut observer que le modèle semble bien plus fiable pour prédire qu'un test sera négatif (20% de faux négatifs) que pour prédire qu'un test sera positif (60% de faux positifs).

   Si on prédit que le test sera positif il faut donc rester très prudent, mais si le modèle prédit un test négatif, il est peu utile de tester quand même.
8. Non puisque la valeur k a été choisie sur la base de ce taux d'erreur, qui est donc nécessairement biaisé puisque cette maximisation du taux gonfle celui-ci. On peut aussi accepter une réponse indiquant qu'une corrélation croisée devrait être réalisée et que se baser uniquement sur le taux de prédiction des 10 premiers n'est pas très fiable.

Exercice 2 Grammaires (type A)

1. Soit $\Sigma_0 = \{\texttt{a}, \texttt{b}, ..., \texttt{z}\}$ l'ensemble des caractères alphanumériques minuscules. Quel est le langage engendré par la grammaire suivante de symbole initial S et d'alphabet terminal Σ_0 ?

$$S \to \varepsilon \,|\, AS \qquad A \to \sigma \text{ avec } \sigma \in \Sigma_0$$

On considère un langage de programmation Ĉ (prononcé *C chapeau*) qui peut être vu comme un sous ensemble de `C`. On rappelle qu'un *identificateur* est un nom choisi par le programmeur qui peut être utilisé pour désigner une variable ou une fonction.

En Ĉ, un identificateur est une séquence de caractères de $\Sigma_1 = \{\texttt{a}, ...\texttt{z}, \texttt{A}, ..., \texttt{Z}, \texttt{0}, ..., \texttt{9}, \texttt{\_}\}$ ne commençant pas par un chiffre. Exemples d'identificateurs : `x` `Tea4Two` `_foo_bar`

2. Donner une grammaire de symbole initial X qui engendre exactement le langage des mots qui sont des identificateurs valides en Ĉ.

Dans toute la suite, on suppose que E est le symbole initial d'une grammaire permettant d'engendrer les expressions en Ĉ. On considère à présent un non-terminal I destiné à engendrer les instructions de Ĉ. Ces dernières sont : soit une expression (éventuellement absente dans le cas d'une instruction vide) suivie du caractère '`;`', soit une instruction `while`, soit un *bloc*. On obtient ainsi les règles suivantes :

(1) $I \longrightarrow E$`;`
(2) $I \longrightarrow$ `;`
(3) $I \longrightarrow$ `while (`E`)` I
(4) $I \longrightarrow B$

Le non-terminal B est destiné à engendrer les blocs de Ĉ, sachant qu'un bloc est une liste (éventuellement vide) d'instructions délimité par une accolade ouvrante et une accolade fermante. Exemple :
`{ while (a!=1){ a= 3*a+1; {} ;; while (a%2 ==0) a= a/2; } }`

3. Montrer que tout mot engendré par I se termine par un point-virgule ou une accolade fermante.

4. Indiquer quelles règles ajouter aux règles (1) à (4) pour que le langage engendré par B soit celui des blocs en Ĉ.

5. Le langage engendré par I est-il rationnel ?

Similairement à `C`, Ĉ a une instruction conditionnelle. Elle est définie syntaxiquement par l'ajout de la règle (5) à I : le non-terminal O désigne une branche sinon (`else`) :

(5) $I \longrightarrow$ `if (`E`)` I O
(6) $O \longrightarrow$ `else` I
(7) $O \longrightarrow \varepsilon$

6. Après avoir ajouté ces règles, la grammaire de symbole initial I est-elle ambiguë ? Justifier votre réponse en considérant l'instruction suivante : `if (x) if (y) a = a+1; else a = a+2;`

Similairement à `C`, la sémantique de Ĉ élimine les ambiguités des `if`/`else` en rattachant le `else` au `if` le plus proche parmi les `if` ambigus. Par exemple, `if (e1) if (e2) e3; else e4;` équivaut sémantiquement à `if (e1) {if (e2) e3; else e4;}`

7. Donner une grammaire non-contextuelle non ambiguë rattachant le `else` au `if` le plus proche.

Proposition de corrigé

1. S engendre $\Sigma_0^\star$.

2. Voici une grammaire convenable (la sémantique de P est "première lettre", celle de M est "mot de $\Sigma_1^\star$" et celle de S est "symbole de Σ_1") :

$$X \to PM \quad P \to \texttt{a}|...|\texttt{z}|\texttt{A}|...|\texttt{Z}|\texttt{\_} \quad M \to \varepsilon|SM \quad S \to P|\texttt{0}|...|\texttt{9}$$

3. Les règles (1), (2) et (4) terminent une dérivation à partir de I par `;` ou `}`. Donc inductivement sur la dérivation (ou par récurrence sur sa longueur), la règle (3) termine aussi une dérivation à partir de I par ces symboles, ce qui conclut.

4. Une possibilité :

$$B \to \texttt{\{}\, L\, \texttt{\}} \quad L \to I\ L \,|\, \varepsilon$$

5. Le langage engendré par I n'est pas régulier. En effet s'il l'était, son intersection avec $\{^\star\}^\star$ le serait. Or ce langage est le langage $\{\{^n\}^n \,|\, n \in \mathbb{N}^\star\}$ qui n'est pas régulier.

6. Cette grammaire est ambiguë car le mot proposé par l'énoncé admet deux arbres de dérivation. Les deux commencent en utilisant la règle (5) puis l'un dérive le O ainsi produit avec la règle (7) alors que l'autre le dérive avec la règle (6).

7. Le if-else est distingué du if sans else : dans le if-else, l'instruction du then peut être tout sauf un if-then. D'où les règles suivantes :

$$I \to \texttt{if (}\, E\, \texttt{)}\, I \,|\, J \qquad J \to E\texttt{;} \,|\, \texttt{;} \,|\, \texttt{while (}\, E\, \texttt{)}\, I \,|\, B \,|\, \texttt{if (}\, E\, \texttt{)}\, J\ \texttt{else}\ I$$

Exercices de type B

Exercice 3 wildcards (type B)

Cet énoncé est accompagné d'un ou plusieurs codes compagnons en OCaml fournissant certaines des fonctions mentionnées dans l'énoncé : ils sont à compléter en y implémentant les fonctions demandées.

On définit un *motif* comme étant un mot sur l'alphabet $\{\texttt{a}, \texttt{b}, \texttt{?}, \texttt{*}\}$ et un *texte* comme étant un mot sur l'alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$. L'objectif de l'exercice est de déterminer si un texte *respecte* un certain motif (on dit alors aussi que le motif *capture* le texte) avec comme conventions que :

- Le caractère `?` dans un motif signifie "n'importe quel caractère de Σ est autorisé".
- Le caractère `*` dans un motif signifie "n'importe quel mot de $\Sigma^\star$ (y compris ε) est autorisé".

Autrement dit, un texte est capturé par un motif si c'est un mot qui appartient au langage décrit par le motif. Par exemple les textes `abba` et `aabaaaba` respectent le motif `a?ba*` (car ils commencent par `a`, puis contiennent une lettre, puis se poursuivent par le facteur `ba`, puis terminent par un nombre quelconque de lettres) alors que ce même motif ne capture pas le texte `abaab`.

1. Parmi les textes suivants, indiquer ceux qui respectent le motif $m_0 =$ `*ab*?a` :

(a) `aab` (c) `abbbbaa` (e) `aaaba`

(b) `abab` (d) `babaa` (f) `abbabba`

Ces textes sont stockés dans une liste `textes` dans le code compagnon ; qui servira ainsi aux tests. Pour les questions 2 à 5, on représente textes et motifs par un type `chaine` défini par :

```
type chaine = char list
```

Remarquons que le mot vide ε est ainsi représenté en OCaml par la liste vide `[]`.

2. Écrire une fonction `chaine_of_string` qui convertit un objet de type `string` en l'objet de type `chaine` correspondant. Ainsi, `chaine_of_string "aa*b"` s'évalue en la liste `['a';'a';'*';'b']`.

3. Vu notre objectif, expliquer pourquoi il est inutile dans un motif d'avoir plusieurs $*$ consécutives.
4. Écrire une fonction `purge : chaine -> chaine` qui supprime les étoiles consécutives inutiles du motif en entrée. Ainsi, `purge ['a';'*';'*';'*';'b';'*']` s'évalue en `['a';'*';'b';'*']`.

On suppose dans la suite qu'on ne travaille qu'avec des motifs purgés, c'est-à-dire sans occurrences consécutives d'étoiles. Au besoin, les motifs seront modifiés via `purge` pour s'en assurer.

Pour savoir si un texte respecte un motif donné, on propose tout d'abord la stratégie récursive naïve suivante. On compare la première lettre u du motif et la première lettre v du texte :

- Si $u \in \{\texttt{a}, \texttt{b}\}$ et coïncide avec v on vérifie que la fin du texte respecte la fin du motif.
- Si $u = \texttt{?}$, on vérifie que la fin du texte respecte la fin du motif.
- Si $u = \texttt{*}$, il y a deux possibilités pour que le texte respecte le motif : soit cette `*` représente ε dans le texte, soit elle représente un mot auquel la lettre v appartient. On vérifie que dans au moins un des deux cas, le morceau de texte restant respecte le morceau de motif restant.

Le code compagnon fournit une fonction (incomplète) `respecte_naif : chaine -> chaine -> bool` prenant dans cet ordre un motif et un texte et destinée à renvoyer `true` si et seulement si le texte respecte le motif en suivant la stratégie précédente.

5. Expliquer brièvement la correction des cas traités dans `respecte_naif`.
6. Compléter la fonction `respecte_naif` et vérifier qu'elle se comporte comme attendu sur les exemples de la question 1.

On admet que la fonction `respecte_naif` a une complexité exponentielle en la taille de son entrée. Pour améliorer cette complexité, on propose de déterminer si un texte $t = t_0 ... t_{|t|-1}$ respecte un motif $m = m_0 ... m_{|m|-1}$ par programmation dynamique.

Pour tout $i \in [\![0, |m|]\!]$ et tout $j \in [\![0, |t|]\!]$, on note $p(i, j)$ le booléen valant vrai si et seulement si le préfixe de longueur j du texte respecte le préfixe de longueur i du motif.

7. Pour $j \in [\![0, |t|]\!]$, que vaut $p(0, j)$? Pour $i \in [\![0, |m|]\!]$, que vaut $p(i, 0)$?

On admet la validité des relations suivantes : pour tout $i \in [\![1, |m|]\!]$ et tout $j \in [\![1, |t|]\!]$ on a :

$$p(i,j) = \begin{cases} p(i-1, j-1) & \text{si } m_{i-1} \text{ vaut ?} \\ p(i-1, j-1) \wedge (m_{i-1} = t_{j-1}) & \text{si } m_{i-1} \text{ vaut a ou b} \\ p(i-1, j) \vee p(i, j-1) & \text{si } m_{i-1} \text{ vaut *} \end{cases}$$

8. Écrire une fonction `respecte_dynamique : string -> string -> bool` prenant en entrée dans cet ordre un motif et un texte sous forme de chaînes de caractères et renvoyant `true` si et seulement si le texte respecte le motif selon la stratégie dynamique. On demande d'adopter une approche de bas en haut (bottom-up). On rappelle l'existence de la fonction `Array.make_matrix`.
9. Déterminer les complexités temporelle et spatiale de `respecte_dynamique`. Peut-on améliorer la complexité spatiale à moindres frais ? Justifier.

Proposition de corrigé

1. Le motif m_0 permet de capturer les textes qui contiennent le facteur ab puis qui contiennent une lettre et terminent par a. Les textes qui respectent m_0 sont c, d, f.
2. On accumule les caractères de l'entrée en tête (pour des raisons de complexité) d'une liste initialement vide. On la renvoie renversée pour respecter l'ordre initial de l'entrée :

```
let chaine_of_string (s:string) :chaine =
  let res = ref [] in
```

```
  for i = 0 to (String.length s) -1 do
    res := s.[i]::(!res)
  done;
  List.rev !res
```

3. La concaténation de k mots quelconques est un mot quelconque. Réciproquement, un mot quelconque m est la concaténation de k mots, par exemple m suivi de $k-1$ fois ε. On en déduit que pour tout $k \geq 1$, le motif $*$ et le motif $\underbrace{*\ldots*}_{k \text{ fois}}$ capturent les mêmes textes.

4. On filtre simplement la chaîne sur ses deux premiers éléments :

```
let rec purge (s:chaine) :chaine =
  match s with
  |[]  |[_] -> s
  |'*'::'*'::q -> purge ('*'::q)
  |a::q -> a::(purge q)
```

5. Le code fourni traite en fait les cas de base :
   - Un texte vide respecte le motif vide (ligne 1).
   - Si le motif est vide mais pas le texte, le texte ne peut pas respecter le motif (ligne 4).
   - Si le texte est vide, le seul motif qui le capture est $*$ (puisque seules les successions de $*$ permettent de capturer ε et que le motif est purgé), d'où les lignes 2 et 3.

6. Il ne reste plus qu'à suivre la stratégie proposée :

```
let rec respecte_naif (motif:chaine) (texte:chaine) :bool =
  match motif, texte with
  |[], [] -> true
  |['*'], [] -> true
  |_, [] -> false
  |[], _ -> false
  |a::b, c::d when a = c || a = '?' -> respecte_naif b d
  |'*'::b, _::d -> (respecte_naif b texte) || (respecte_naif motif d)
  |_ -> false
```

   Le dernier cas permet de traiter la situation où le motif commence par une lettre de $\{a, b\}$ mais qui ne coïncide pas avec la première lettre du texte. L'avant-dernier cas traduit l'énoncé. Premier cas : on ne consomme pas de lettre du texte, ce qui correspond au cas où $*$ représente ε dans le motif. Second cas : l'$*$ du motif consomme la première lettre du texte ; et est susceptible de consommer les suivantes.

   On vérifie ensuite les réponses de la question 1 :

```
let m0 = chaine_of_string "*ab*a?"
let _ = List.map (respecte_naif m0) (List.map chaine_of_string textes)
```

7. On constate que répondre à ces questions revient à traduire les cas de base de la fonction `respecte_naif` :
   - $p(0,0)$ vaut vrai puisque ε respecte ε.
   - Pour tout $j \geq 1$, $p(0,j)$ vaut faux car un texte non vide ne peut pas respecter ε.
   - Pour tout $i \geq 2$, $p(i,0)$ vaut faux car ε ne peut pas respecter un motif de taille au moins deux ; ce dernier contenant au moins un symbole différent de $*$ puisqu'il est purgé.

- Enfin il reste le cas particulier de $p(1,0)$, qui vaut vrai si et seulement si la première lettre du motif est $*$ car c'est la seule possibilité pour que ε le respecte.

8. On souhaite calculer $p(|m|,|t|)$ puisque c'est le booléen valant vrai si et seulement si le préfixe de taille le texte du texte (donc le texte) respecte le préfixe de taille le motif du motif (donc le motif).

   On calcule donc la matrice $(p(i,j))_{0\leq i\leq |m|, 0\leq j\leq |t|}$, de taille $(1+|m|)\times(1+|t|)$. Sa première ligne et sa première colonne se remplissent grâce à la question 6. Puis, on la remplit ligne à ligne (ce qui est possible vu les relations). Il ne reste plus qu'à extraire le bon coefficient :

```
let respecte_dynamique (motif:string) (texte:string) :bool =
  let n = String.length motif in
  let m = String.length texte in
  let res = Array.make_matrix (n+1) (m+1) false in
  res.(0).(0) <- true;
  res.(1).(0) <- motif.[0] = '*';
  for i = 1 to n do
    for j = 1 to m do
      match motif.[i-1] with
      |'?' -> res.(i).(j) <- res.(i-1).(j-1)
      |'*' -> res.(i).(j) <- res.(i-1).(j) || res.(i).(j-1)
      |_ -> res.(i).(j) <- res.(i-1).(j-1) && motif.[i-1] = texte.[j-1]
    done;
  done;
  res.(n).(m)
```

9. Calculer un coefficient de la matrice précédente se fait en temps constant. On en déduit que la complexité temporelle de `respecte_dynamique` est en $O(|t||m|)$. C'est aussi le cas de la complexité spatiale.

   On peut la faire passer à un $O(|t|)$ en ne conservant que deux lignes de la matrice à tout moment et même à un $O(\min(|m|,|t|))$ car la relation de récurrence permet un remplissage ligne à ligne ou colonne par colonne : on peut donc choisir ce qui est le plus avantageux du point de vue spatial.

Exercice 4 Saisie swype (type B)

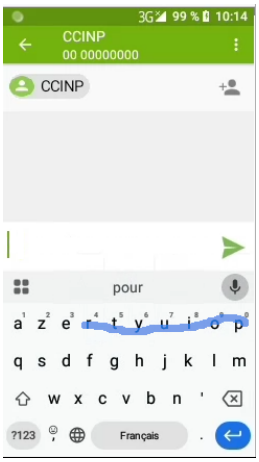
Cet énoncé est accompagné d'un ou plusieurs codes compagnons en C fournissant certaines des fonctions mentionnées dans l'énoncé : il sont à compléter en y implémentant les fonctions demandées.

La ligne de compilation `gcc -o main.exe -Wall *.c -lm` *vous permet de créer un exécutable* `main.exe` *à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire* `make`*. En ligne de commande, il suffit d'écrire* `make`*. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* `./main.exe`*.*

Il est possible d'activer davantage d'avertissements et un outil d'analyse de la gestion de la mémoire avec la ligne de compilation `gcc -o main.exe -g -Wall -Wextra -fsanitize=address *.c -lm` *ou en écrivant* `make safe`*. L'examinateur pourra vous demander de compiler avec ces options.*

Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant `make clean` *et relancer une compilation.*

Certains claviers de smartphones autorisent une saisie par glissement du doigt (*swype* en anglais) : l'utilisateur pose son doigt sur la première lettre du mot et, sans soulever le doigt, le fait glisser de sorte à ce qu'il passe successivement et dans l'ordre sur les lettres du mot qu'il souhaite écrire, jusqu'à la dernière lettre de ce mot. À ce moment-là, il peut soulever son doigt et un mot apparaît à l'écran, qui appartient à la langue dans laquelle le clavier est paramétré. Bien entendu, pendant le déplacement du doigt, ce dernier est passé sur des caractères qui n'appartiennent pas au mot. Par exemple, sur un clavier azerty réglé en français, pour écrire le mot "pour", il suffit de faire glisser horizontalement le doigt de la lettre "p" à la lettre "r". On passe par les lettres "p", "o", "i", "u", "y", "t" et "r", dont certaines appartiennent au mot "pour" et d'autres non, comme illustré dans la figure ci-contre.



Le but de cet exercice est de trouver tous les mots d'un lexique qui sont sous-mots d'une suite de lettres saisie et possèdent les mêmes première et dernière lettre que cette suite.

Dans tout le sujet, on suppose que la langue de paramétrage du sujet est le français, et on ne considère que des mots écrits avec des lettres latines minuscules sans accents, à l'exclusion de tout autre type de caractères (par exemple pas de tiret ou d'apostrophe dans les mots considérés).

Le code fourni est mis dans trois fichiers :

- le fichier d'en-tête `auxiliaire.h` contient les déclarations des types et fonctions, et la documentation associée : il faut le lire ;
- le fichier `auxiliaire.c` contient les définitions des fonctions fournies ; il est inutile de le regarder, mais il est nécessaire pour la compilation ;
- le fichier `compagnon.c` qui contient une fonction `main` et des fonctions à trou pour certaines questions ; c'est dans ce fichier que vous devez écrire votre code.

Vous est par ailleurs fourni un fichier `lexique.txt` contenant un lexique (le code compagnon se sert de ce fichier pour remplir une certaine structure de données décrite en partie 2, inutile pour vous de connaître le format de ce fichier).

1 Fonctions auxiliaires

Pour des raisons pratiques sur lesquelles nous reviendrons plus tard, pendant l'exécution de l'algorithme principal, les mots sont stockés à l'envers (de la dernière à la première lettre).

Un mot est représenté en `C` par une chaîne de caractères, et nous représenterons un ensemble de mots par une liste chaînée de type `liste` qui est un synonyme de `struct maillon *`, types définis dans

`auxiliaire.h` (sans nous soucier des éventuels doublons).

1.1 Affichages

1. Écrire une fonction `void affiche_miroir(char *mot)` qui affiche une chaîne de caractères à l'envers (de la dernière à la première lettre).
2. Écrire une fonction `void affiche_liste(liste lst)` qui affiche tous les mots de la liste `lst` séparés par des retours à la ligne, chaque mot étant affiché à l'envers.

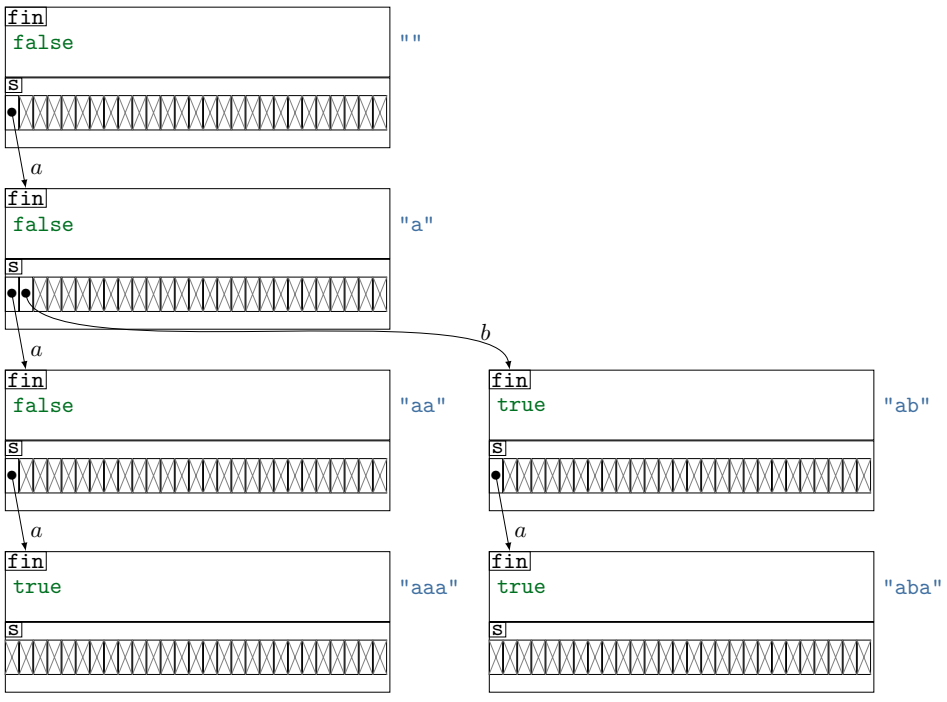
1.2 Gestion de listes de mots

Notez que les maillons des listes contiennent chacun un mot (champ `mot`) et sa longueur (champ `n`).

3. Compléter la fonction `void ajout_lettre(char c, liste lst)` qui ajoute le caractère `c` à la fin de chaque chaîne de caractère de la liste `lst` (ce qui correspond donc à un ajout au début de chaque mot lors de l'affichage). On suppose (sans avoir à le vérifier) que la place mémoire allouée pour chaque champ `mot` est suffisante pour cet ajout. La complexité de votre fonction doit être linéaire en la longueur de la liste (sans avoir à le justifier).
4. Justifier brièvement la complexité temporelle de votre fonction dans le pire des cas. Quelle aurait été la complexité d'une fonction ajoutant le caractère `c` au début de chaque mot de la liste `lst` ?

2 Fonction principale : Recherche de mots

Le fichier d'en-tête `auxiliaire.h` vous fournit un type `lexique` et une fonction `genere_lexique` permettant d'obtenir une adresse d'une valeur de ce type (cette fonction utilise le fichier `lexique.txt` également fourni). Le type `lexique` représente un arbre lexicographique dont chaque nœud interne contient 26 fils rangés dans un tableau : le fils d'indice `0` représente la lettre a, le fils d'indice `1` le lettre b, etc. Chaque nœud de cet arbre contient un marqueur (le champ `fin`) qui indique si la suite des lettres de la racine à ce nœud est un mot du lexique ou pas. Si un fils est `NULL`, cela signifie que le mot atteint suivi de la lettre correspondant au fils n'est pas le préfixe d'un mot existant dans le lexique. La figure ci-dessous donne l'arbre lexicographique obtenu pour le lexique constitué des mots `"aaa"`, `"ab"` et `"aba"`. À droite de chaque nœud de l'arbre est indiquée l'étiquette du chemin de la racine vers ce nœud.



Le fichier d'en-tête `auxiliaire.h` vous fournit une fonction `int num(char c)` qui à partir d'un caractère supposé être une lettre minuscule sans accent fournit l'indice associé à cette lettre dans l'alphabet.

Soit deux mots finis u et v sur un alphabet donné, tels que v est de longueur au moins 2. On dit que u est un sous-mot *final* de v si c'est un sous-mot de v et que u et v se terminent par la même lettre. On dit que u est un sous-mot *admissible* de v si c'est un sous-mot final de v et que u et v commencent par la même lettre.

Pour un mot v fixé de longueur au moins 2, on cherche à déterminer les mots d'un lexique qui sont des sous-mots admissibles de v. On note (P) ce problème.

5. Justifier brièvement que le nombre de sous-mots distincts d'un mot donné est au plus exponentiel en la longueur du mot. Pourquoi cela peut-il être moins ?

Nous allons utiliser une technique de retour sur trace (*backtracking*) pour résoudre le problème (P) : l'idée est de parcourir le mot v et l'arbre représentant le lexique en même temps : pour chaque lettre de v, à l'exclusion des première et dernière, on teste la possibilité de conserver ou non cette lettre. Chaque choix fait aboutit alors à un ensemble de mots possibles, et on considère l'union de ces deux ensembles. Pour simplifier, on autorise les doublons.

6. On suppose que la longueur d'un mot du lexique est majorée par la constante `LG_MAX` donnée dans `auxiliaire.h` *et on rappelle qu'il est possible d'obtenir l'adresse de la case d'indice `i` d'un tableau `t` avec la syntaxe `&t[i]` (note du jury : rappel non présent dans l'énoncé original que le jury a souhaité rajouter à la publication)*.

   Compléter la fonction `liste *sous_mots_possibles(lexique *lex, char *mot, bool obligatoire)` qui prend en argument un arbre représentant un lexique, une chaîne de caractères et un marqueur spécifiant s'il faut garder la première lettre de la chaîne et renvoie une liste contenant tous les mots du lexique qui sont des sous-mots finaux de `mot` si `obligatoire` vaut `false` et qui sont des sous-mots admissibles de `mot` si `obligatoire` vaut `true`. :

   (a) cas où le mot est vide (section `A COMPLETER (a)`) : si la racine du lexique correspond à une fin de mot, alors `res` doit être une liste contenant un unique maillon dont le champ `mot` permet de stocker une chaîne de caractère de longueur `LG_MAX` et est une chaîne de caractères vide ;

   (b) cas où on garde la première lettre du mot courant (section `A COMPLETER (b)`) : `lst1` doit contenir les sous-mots de `mot` qui contiennent sa première lettre et appartiennent au lexique, stockés à l'envers (pour les raisons de complexité vues en question 4) ;

   (c) cas où on ne garde pas la première lettre du mot courant (section `A COMPLETER (c)`) : `lst2` doit contenir les sous-mots de $v_2 \cdots v_k$ où `mot`$=v_1 \cdots v_k$ (c'est-à-dire les sous-mots de `mot` privé de sa première lettre) et appartenant au lexique, stockés à l'envers (pour les raisons de complexité vues en question 4).

7. Prouver la correction totale de l'algorithme précédent.

Proposition de corrigé

1.

```c
void affiche_miroir(char *mot){  // Q1
  for(int i=strlen(mot)-1; i>=0; i=i-1){
    printf("%c", mot[i]);
  }
}
```

2.

```c
void affiche_liste(liste lst){ // Q2
  while(lst != NULL){
    affiche_miroir(lst->mot);
    printf("\n");
    lst = lst->suite;
  }
}
```

3.

```c
void ajout_lettre(char c, liste lst){  // Q3
  for(liste m=lst; m != NULL; m=m->suite){
    m->mot[m->n] = c;
    m->mot[m->n+1] = '\0';
    m->n = m->n + 1;
  }
}
```

4. Chaque itération se fait en temps constant, il y a autant d'itérations que de mots dans la liste, la complexité temporelle est donc linéaire en la longueur de la liste.

   Si l'ajout se fait en début de mot, il faut décaler les autres lettres, chaque itération se ferait en temps linéaire en la longueur du suffixe, qui est majorée par la longueur du plus long mot. Si tous les mots ont même longueur, ce temps est atteint à chaque itération. Le nombre d'itérations n'est pas modifié. La complexité est donc linéaire en le produit de la longueur de la liste et de la longueur du plus long mot de cette liste. On peut aussi sans la dernière hypothèse simplificatrice majorer par la somme des longueurs des mots.

5. Soit un mot v de longueur n indicé à partir de 0. Choisir un sous-mot de v, c'est choisir une fonction $[\![0, n-1]\!] \to \{V, F\}$. Il y a donc 2^n choix possibles. Ces choix n'aboutissent pas toujours à des mots distincts si les lettres de v ne sont pas deux à deux distinctes. Par exemple si $v = a^n$, v possède $n+1$ sous-mots distincts.

6.

```c
liste sous_mots_possibles(lexique *lex, char *mot, bool obligatoire){  // Q6
  if(lex == NULL){
    return NULL;
  }
  if(*mot == '\0'){
    if(!lex->fin){ //on n'est pas sur un mot du lexique
      return NULL;
    } else {
      liste res = NULL;
      // A COMPLETER (a) - debut
      res = malloc(sizeof(struct maillon));
      res->mot = malloc((LG_MAX+1)*sizeof(char));
      res->mot[0] = '\0';
      res->n = 0;
      res->suite = NULL;
      // A COMPLETER (a) - fin
      return res;
    }
  }
  liste lst1 = NULL;
```

```
21    // A COMPLETER (b) - debut
22    int nc = num(mot[0]);
23    lst1 = sous_mots_possibles(lex->s[nc], &mot[1], false);
24    ajout_lettre(mot[0], lst1);
25    // A COMPLETER (b) - fin
26    liste lst2 = NULL;
27    obligatoire = obligatoire || mot[1] == '\0'; //dernière lettre obligatoire
28    if(!obligatoire){ //lettre pas obligatoire : on la saute
29      // A COMPLETER (c) - debut
30      lst2 = sous_mots_possibles(lex, &mot[1], false);
31      // A COMPLETER (c) - fin
32    }
33    return union_listes(lst1, lst2);
34  }
```

7. Par récurrence sur la longueur du sous-mot.

   On suppose `lex` non vide, sinon la terminaison et la correction sont immédiates.

   cas de base si `mot` est le mot vide :

   - si la racine de `lex` n'est pas finale, alors l'appel se termine avec la bonne valeur en ligne 7 ;
   - sinon un maillon est créé contenant le mot vide (lignes 11 à 15) et l'appel se termine avec la bonne valeur en ligne 17.

   hérédité Supposons que tout appel à `sous_motes_possibles` se termine et soit correct, pour toute chaîne de caractères `mot` de longueur n pour un certain $n \geq 0$. Soit une chaîne de caractères `mot` de longueur $n+1$ (qu'on suppose inférieure à `LG_MAX`). L'ensemble des sous-mots de `mot` s'écrit comme une partition entre :

   - l'ensemble de ses sous-mots qui contiennent la première lettre de `mot` : obtenu en temps fini avec les lignes 23 et 24 par hypothèse de récurrence ;
   - l'ensemble de ses sous-mots qui ne contiennent pas la première lettre de `mot` si elle n'est pas obligatoire : obtenu en temps fini avec les lignes 26 et 30 par hypothèse de récurrence.

   Le retour à la ligne 33 donne donc bien le résultat attendu en temps fini.