Ce sujet est adapté des épreuves d'informatique issues des concours (Centrale-Supélec, MP, 2009) et (CCP, MP, 2019).

Les deux parties sont indépendantes.

I Partie 1 : Chasse aux fantômes

Un groupe de n chasseurs de fantômes combat un groupe de n fantômes dans une grotte dont la base est un disque. Chaque chasseur est armé d'un canon à ions, capable d'éradiquer un fantôme d'un coup de rayon. Un rayon se propage en ligne droite et termine sa course en touchant le fantôme. Les chasseurs sont confrontés à deux problèmes : il est nécessaire d'éliminer tous les fantômes en même temps, sinon ils se multiplient pour revenir au nombre initial de n fantômes ; il est très dangereux que deux rayons se croisent, cela ne doit donc pas se produire.

Les $2n$ protagonistes sont représentés par des points distincts $P_1, \cdots, P_{2n}$ répartis dans l'ordre des indices croissants sur la circonférence d'un cercle parcouru dans le sens trigonométrique.

Dans une première sous-partie, on s'intéresse à la notion de *stratégie valide* et au test (efficace) de validité d'une stratégie. Dans une seconde, on s'attache à trouver efficacement des stratégies gagnantes pour les chasseurs de fantômes.

I.1 Stratégie valide

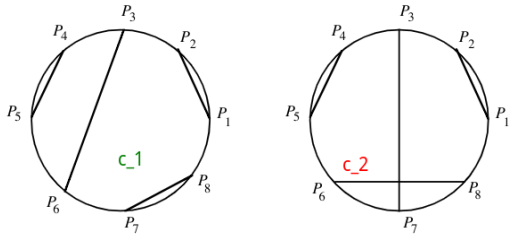
Dans cette partie, on cherche les liens possibles entre les P_i, sans s'occuper de qui est chasseur et qui est fantôme. Les points P_i doivent être reliés deux à deux par une corde du cercle sans que celles-ci ne se croisent.

Dans toute la suite, on fixe un entier $n > 0$ et on appelle *stratégie de taille* n un vecteur $(c[1], \cdots, c[2n])$ tel que pour tout entier i vérifiant $1 \leqslant i \leqslant 2n$ on ait:

$$1 \leqslant c[i] \leqslant 2n, \qquad c[c[i]] = i, \qquad c[i] \neq i.$$

L'ensemble des cordes décrites par le vecteurs est $\{(i, c[i])\}_i$. Cette stratégie est dite *admissible* si les segments ainsi formés ne se coupent pas. On admettra que cette propriété ne dépend pas de la position des points sur la circonférence mais seulement de la stratégie.
En `Python`, on représente ce vecteur par un tableau de taille $2n + 1$ dont le premier élement ne sera pas utilisé, de sorte que les points soient bien aux indices 1 à $2n$ dans le vecteur `Python`.

Par exemple, la stratégie admissible `c_1 = [0, 2, 1, 6, 5, 4, 3, 8, 7]` et la stratégie non-admissible `c_2 = [0, 2, 1, 7, 5, 4, 8, 3, 6]` sont représentées ci-après.



▸ **Question 1.** *Dans le cas $n = 3$, donner le nombre de stratégies possibles, admissibles ou non.*

▸ **Question 2.** *Préciser le nombre de stratégies de taille 3 qui sont admissibles. Les représenter sur une figure.*

▸ **Question 3.** *Déterminer le nombre de stratégies de taille n entier positif (qu'elles soient ou non admissibles).*

I.1.1 Un test naïf d'admissibilité

Soit i, j, k, l quatre entiers distincts tels que $1 \leqslant i < j \leqslant 2n$ et $1 \leqslant k < l \leqslant 2n$.

▸ **Question 4.** *Donner une condition nécessaire et suffisante, portant uniquement sur ces entiers, pour que les segments $[P_i, P_j]$ et $[P_k, P_l]$ se croisent. Écrire une fonction* `croise(i,j,k,l) -> Bool` *basée sur cette condition. On supposera que les conditions sur les entiers, spécificiées plus haut, sont respectées.*

▸ **Question 5.** *En déduire une fonction* `estAdmissible(c) -> Bool` *qui prend en paramètre le vecteur de stratégie et renvoie un booléen dont le résultat est* `True` *si la stratégie est admissible.*

▸ **Question 6.** *Estimer, en la justifiant, la complexité de cette méthode.*

I.1.2 Un test d'admissibilité plus efficace

On souhaite tester la stratégie de façon plus efficace. Pour cela, on utilise une fonction `evalue` qui prend deux entiers i et j comme paramètres (ces derniers étant compris entre 1 et $2n$) et dont le résultat est `true` si et seulement si les deux propositions suivantes sont vraies.

1. $(i \leqslant k \leqslant j) \Rightarrow (i \leqslant c[k] \leqslant j)$
2. Les segments ayant leurs extrémités dans l'arc (fermé) $[P_i, P_j]$ ne se croisent pas.

▸ **Question 7.** *Donner la valeur de* `evalue(i, j)` *pour chacun des cas particuliers suivants:*

a) $j < i$; *b)* $j \geqslant i$ *et* $c[i] < i$; *c)* $j \geqslant i$ *et* $c[i] > j$.

▸ **Question 8.** *Pour $i < c[i] < j$, montrer que* `evalue(i,j)` *se déduit de* `evalue(i+1, c[i]-1)` *et de* `evalue(c[i]+1, j)`.

] n'en croise aucun autre. Puisque $i < c[i] < j$, `evalue(i,j)` vaut `True` également.
Finalement,

```
evalue(i,j) = evalue(i+1, c[i]-1) and evalue(c[i]+1, j)
```

▸ **Question 9.** *Écrire une fonction* `testStrategie` *qui prend en paramètre un vecteur de stratégie et renvoie un booléen dont la valeur est* `True` *si la strategie est admissible, de complexité $O(n)$. Comment s'appelle la méthode algorithmique que vous avez utilisé ?*

▸ **Question 10.** *Démontrer avec soin que la fonction* `testStrategie` *est de complexité $O(n)$.*

I.2 Retour au problème initial

La méthode précédente teste si une stratégie est admissible. On souhaite maintenant, le statut (chasseur ou fantôme) des points étant donné, déterminer directement une stratégie admissible, de façon également efficace.

▸ **Question 11.** *Montrer que si P_i est un chasseur, alors il existe $j \neq i$ tel que P_j est un fantôme et tel que le nombre de chasseurs soit égal au nombre de fantômes de chaque côté de l'axe (P_i, P_j).*

▸ **Question 12.** *En déduire un algorithme simple pour construire un stratégie admissible. Comment s'appelle la technique algorithmique que vous avez utilisé ?*

▸ **Question 13.** *Déterminer la complexité de l'algorithme précédent dans le pire cas et évaluer, sans démonstration, sa complexité moyenne.*

On considère que le problème est écrit par un vecteur `p` qui contient la nature des points P_i: `p[i]` vaut 1 pour un chasseur, et `p[i]` vaut -1 pour un fantôme. En particulier, `sum(p) == 0` puisqu'il y a autant de chasseurs que de fantômes.

▸ **Question 14.** *Écrire une fonction* `cibles`*, qui prend en paramètre le vecteur p et renvoie la liste des indices (i, j) – où P_i est un chasseur, et P_j un fantôme – composant une stratégie admissible du problème.*

II Partie 2 : Algorithmes de compression

Nous proposons dans cette partie d'étudier une méthode de compression de données. L'algorithme proposé ici implémente plusieurs couches d'arrangement de données et de compression successives, utilisées dans l'ordre suivant pour la compression et l'ordre inverse pour la décompression : [1] Transformation de Burrows-Wheeler (BWT), [2] Codage par plages (RLE) et [3] Codage de Huffman.

▸ **Question 15.** *Motiver, en quelques lignes et comme vous le feriez pour vos élèves, l'utilisation des algorithmes de compression de données. Vous donnerez un exemple d'utilisation ayant un impact positif sur l'environnement.*

Soit Σ un alphabet de symboles, de cardinal $|\Sigma| = h$ et muni d'une relation d'ordre $\leqslant$. On note Σ^k l'ensemble des mots de longueur k construites à partir de Σ ; il est muni d'une relation d'ordre lexicographique induite par la relation d'ordre $\leqslant$.
Pour $m \in \Sigma^k$, on note $|m| = k$ la taille de m, et $|m|_a$ le nombre d'occurences de $a \in \Sigma$ dans m.

Dans toute cette partie, lorsqu'il s'agira de coder une fonction `Python`, un mot $m \in \Sigma^k$ sera représenté par une chaîne de caractères en `Python` (`string`).

Sans spécification contraire, la *complexité* renvoie à la complexité temporelle.

Les trois sous-parties qui suivent sont indépendantes.

II.1 Transformation de Burrows-Wheeler (BWT)

Soit $m \in \Sigma^k$ un mot. La transformation BWT réalise une permutation des symboles de m de sorte que les symboles identiques sont regroupés dans de longues séquences. Cette transformation n'effectue pas de compression, mais prépare donc à une compression plus efficace.
Dans la suite, nous étudions le codage et le décodage d'un mot transformé par cette opération.

II.1.1 Phase de codage

On ajoute à la fin de m un marqueur de fin, appelé *sentinelle*. Il est noté # par convention, et est inférieur (au sens de $\leqslant$) à tous les autres symboles de Σ. Dans la suite, on désignera par $\widehat{m}$ le mot m auquel on a ajouté le symbole #.

▸ **Question 16.** *Pour m = turlututu, contruire une matrice $\boldsymbol{M}$ dont les lignes sont les différentes permutations circulaires successives du mot $\widehat{m}$. Les permutations seront ici envisagées par décalage à droite des caractères.*

▸ **Question 17.** *Écrire en `Python` une fonction `matrice_mot` qui construit la matrice $\boldsymbol{M}$ à partir d'un mot passé en entrée. La valeur de retour est une liste de chaines de caractères. Cette fonction s'appuiera sur une fonction `perm_circ_d`, que vous définirez, qui réalise une permutation à droite d'un mot m donné en entrée. Par exemple, `perm_circ_d("abc")` renvoie `"cab"`.*

Les lignes de $\boldsymbol{M}$ sont alors permutées de sorte à classer les lignes par ordre lexicographique. On note $\boldsymbol{M'} = \boldsymbol{PM}$ la matrice obtenue, $\boldsymbol{P}$ étant la matrice de permutation.

▸ **Question 18.** *Donner les matrices $\boldsymbol{P}$ et $\boldsymbol{M'}$ dans le cas du mot m = turlututu.*

Pour constuire la matrice de permutation $\boldsymbol{P}$, il faut trier la liste des mots définissant $\boldsymbol{M}$. La méthode de tri choisie ici est le *tri par insertion*.

▸ **Question 19.** *Écrire une fonction Python `tri_par_insertion(liste)` qui réalise le tri par insertion d'une liste d'éléments.*

▸ **Question 20.** *On suppose une liste l composées d'éléments tous distincts (au nombre de n). Soit l' la liste obtenue après tri de l. Combien de listes différents aurait aussi donné l' après tri ? Proposer un code pour évaluer ce nombre pour grand ($n > 1000$) après avoir souligné le problème que cela soulevait.*

▸ **Question 21.** *En déduire une fonction* `matrice_mot_triee` *qui construit $\boldsymbol{M'}$ à partir de $\boldsymbol{M}$. On rappelle qu'en Python la fonction* `<=` *définie sur les listes repose sur l'ordre lexicographie.*

▸ **Question 22.** *Pour $\widehat{m} \in \Sigma^k \times \{\#\}$, donner le nombre de comparaisons de symboles nécessaires dans le pire des cas, pour trier deux permutations circulaires du mot $\widehat{m}$. En déduire un majorant ($O(\cdot)$) de la complexité dans le pire des cas pour le tri des $k+1$ permutations circulaires d'un mot $\widehat{m} \in \Sigma^k \times \{\#\}$ (exprimée en nombre de comparaison de symboles).*

▸ **Question 23.** *Un élève vous demande si on aurait pû utiliser le tri fusion (vu la semaine passée) pour améliorer la complexité de cette étape. Que lui répondez-vous ?*

La transformation de Burrows-Wheeler consiste alors à coder le mot m par la dernière colonne de la matrice $\boldsymbol{M'}$ obtenue à l'aide de $\widehat{m}$. On note $\mathsf{BWT}(m)$ ce codage.

▸ **Question 24.** *Écrire une fonction* `codageBWT` *qui encode un mot passé en entrée. Donner le codage du mot $m = turlututu$.*

II.1.2 Phase de décodage

Pour décoder un mot codé par BWT, il est nécessaire de reconstruire itérativement $\boldsymbol{M'}$ à partir de la seule donnée du mot codé $\mathsf{BWT}(m)$. Par construction, $\mathsf{BWT}(m)$ est la dernière colonne de $\boldsymbol{M'}$. On pose ici comme exemple $\mathsf{BWT}(m) = ednngvnea\#$.

▸ **Question 25.** *Construire, à partir de la seule donnée de $BWT(m)$, la première colonne de $\boldsymbol{M'}$. Justifier le principe de construction.*

La dernière et la première colonne de $\boldsymbol{M'}$ donnent alors tous les sous-mots de longueur 2 du mot $\widehat{m}$.

▸ **Question 26.** *Proposer un algorithme permettant d'obetnir la deuxième colonne de $\boldsymbol{M''}$. Donner cette deuxième colonne pour $BWT(m) = ednngvnea\#$.*

▸ **Question 27.** *On dispose à l'itération n des $(n-1)$ premières colonnes de $\boldsymbol{M'}$ et de sa dernière colonne. Proposer un principe algorithmique permettant de construire la n-ième colonne de $\boldsymbol{M'}$.*

▸ **Question 28.** *En déduire un algorithme itératif permettant de reconstruire $\boldsymbol{M'''}$. Quel décodage obtient-on pour le mot $BWT(m)$ proposé ?*

II.2 Codage par plages RLE

Le codage RLE (Run Length Coding), ou codage par plages, est une méthode de compression dont le principe est de remplacer dans une chaîne de symboles une sous-chaîne de symboles identiques par le couple constitué du nombre de symboles identiques et du symbole lui-même. Par exemple, la chaîne `"aaababb"` est compressée en `[(3, 'a'), (1, 'b'), (1, 'a'), (2, 'b')]`.

▸ **Question 29.** *Proposer, comme vous le feriez face à vos élèves, un type* `Python` *pour la compression RLE qui permet de représenter le résultat comme indiqué précédemment. Vous soulignerez son aspect naturel.*

▸ **Question 30.** *Rédiger un court énoncé à destination de vos élèves autour de ce codage. Vous proposerez une correction des fonctions* `Python` *que vous demanderez d'implémenter, telle que vous la proposeriez à vos élèves.*

II.3 Codage de Huffman

La dernière étape de l'algorithme proposé implémente le codage de Huffman, qui utilise la structure d'arbre binaire. Le principe est de coder un symbole de manière d'autant plus courte que son nombre d'occurrences dans le mot est élevé. L'arbre de Huffman se construit à l'aide de l'algorithme suivant.

Algorithme 2: Codage de Huffman

Entrée : μ un mot de taille $|\mu|$
Sortie : `Huffman`(μ) le codage de Huffman de μ
pour $a \in \Sigma$ **faire**
  si $|\mu|_a > 0$ **alors**
    créer un noeud $(a, |\mu|_a)$
$\mathcal{L} \leftarrow$ liste des noeuds dans l'ordre croissant des poids
$\mathcal{A} \leftarrow$ liste vide
tant que *(longueur($\mathcal{L}$)+longueur ($\mathcal{A}$)>1)* **faire**
  $(g, d) \leftarrow$ deux noeuds de plus faible poids parmi les 2 premiers noeuds de $\mathcal{L}$ et les 2 premiers noeuds de $\mathcal{A}$
  Créer un noeud t
  $n_t \leftarrow n_g + n_d$
  $gauche(t) \leftarrow g$
  Coder la branche de t à g par 0
  $droite(t) \leftarrow d$
  Coder la branche de t à d par 1
  Insérer t à la fin de $\mathcal{A}$
  Retirer g et d de $\mathcal{L}$ ou de $\mathcal{A}$
`Huffman`(μ)$\leftarrow \mathcal{A}$

▸ **Question 31.** *Constuire le codage de Huffman du mot $m =$ "turlututu" en utilisant l'algorithme ci-dessus. Vous expliciterez par des dessins d'arbres chacun des étapes de construction de* `Huffman(m)`.

▸ **Question 32.** *Quelle est la forme de l'arbre de Huffman dans un mot où tous les symboles ont le même nombre d'occurences ?*

▸ **Question 33.** *Vous avez réalisé une activité sur le codage de Huffman avec vos terminales NSI (structure d'arbre binaire, opérations internes de Huffman, Huffman et exemples sommaires). Une élève finit en avance. Proposer une extension de l'activité autour la complexité que représente le calcul de $|m|_a$, pour $a \in \Sigma$, en particulier pour des grands textes.*

••• FIN •••