ACADÉMIE
DE NORMANDIE
Liberté
Égalité
Fraternité

aefe
Agence pour l'enseignement français à l'étranger

Olympiades d'informatique 2024

- Classe de première -

L'épreuve - d'une durée de trois heures - se déroule par équipe et sur ordinateur.

Le sujet est constitué de deux parties indépendantes qui peuvent être traitées dans l'ordre souhaité. Un dossier est mis à disposition des équipes avec les fichiers requis pour certaines questions.

Le dossier intitulé **SUJET** contenant les fichiers requis pour traiter les questions sera à mettre à disposition des équipes.

Les fichiers créés par les candidats seront à joindre dans un unique dossier compressé et qui seront nommés de la façon suivante :

NomEtablissement_VilleEtablissement_EquipeXX_Squelette.png
NomEtablissement_VilleEtablissement_EquipeXX_minuties.png
NomEtablissement_VilleEtablissement_EquipeXX_parodin.py
NomEtablissement_VilleEtablissement_EquipeXX_equipe.odt

où le numéro de l'équipe sera attribué par vos soins.

Le quatrième fichier (equipe.odt) recense les noms des membres de l'équipe.

Les calculatrices sont autorisées selon la réglementation en vigueur.

Les démarches, même incomplètes seront prises en compte dans l'évaluation.

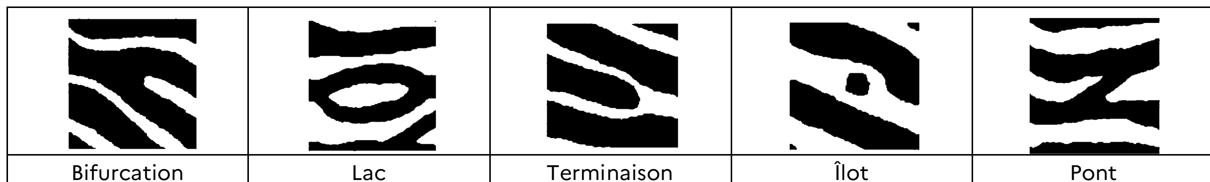
Sif
Société informatique de France

UFR SCIENCES & TECHNIQUES

GREYC
Laboratoire de recherche en sciences du numérique

Traitement et analyse d’une empreinte digitale

Les empreintes digitales sont propres à chaque individu (elles sont même distinctes chez les jumeaux). L’analyse de leurs images à des fins de reconnaissance, que ce soit pour des enquêtes judiciaires ou plus simplement sur des systèmes de sécurité (comme par exemple dans certains smartphones, certaines serrures biométriques ou pour l’authentification de pièces d’identités), s’appuie sur l’examen d’éléments distinctifs appelées *minuties* :



Bifurcation	Lac	Terminaison	Îlot	Pont

Quelques exemples de minuties

L’un des procédés permettant de les recenser consiste en l’application de plusieurs étapes de traitement de l’image numérisée de l’empreinte digitale qui peuvent se résumer à :

- La ***binarisation*** (obtention d’une image constituée uniquement de pixels noirs ou blancs)

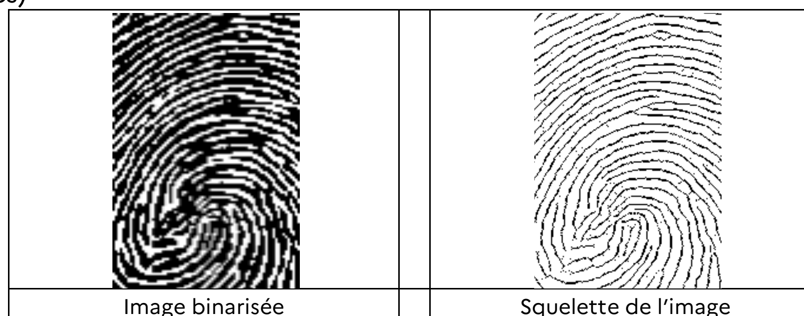


Image binarisée		Squelette de l’image

- La ***squelettisation*** de l’empreinte, qui consiste à réduire l’épaisseur de chacune des lignes de l’empreinte binarisée à un pixel conduisant à un *squelette* de l’image.
- Repérer un ensemble de minuties.

L’objectif de cet exercice est, à partir d’une image binarisée en noir et blanc d’une empreinte digitale (`Empreinte_digitale.png`), d’obtenir l’image en noir et blanc de son squelette (que l’on nommera `squelette.png`)

Rappels sur les pixels et la bibliothèque PIL

On rappelle que la couleur d’un pixel est représentée par trois entiers variant entre 0 et 255, qui constituent la quantité de rouge, de vert et de bleu. Une image est en niveaux de gris lorsque n’importe lequel de ses pixels a la même quantité de rouge que de vert et de bleu.
On rappelle également que la couleur noire correspond aux valeurs (0, 0, 0).
Pour la réalisation de votre code, vous allez utiliser la bibliothèque PIL, dont voici un usage possible (le fichier Python correspondant devant se trouver dans le même dossier que l’image image1.png) :

```
from PIL import Image
img1 = Image.open("image1.png")
img2 = Image.new("RGB", img1.size)
```

```
largeur, hauteur = img1.size
img2.save("image_modifiee.png")
```

La variable `img2` représente une image, pour l'instant composée uniquement de pixels noirs, aux mêmes dimensions que l'image fournie.
Votre code utilisera `getpixel` et `putpixel` de la bibliothèque PIL.

Par exemple, le code :

```
(rouge, vert, bleu) = img1.getpixel((0,0))
img2.putpixel((0,0), (rouge, vert, bleu))
```

permet de récupérer les quantités de rouge, vert, bleu du pixel de coordonnées (0,0) de la première image et de colorer le pixel correspondant de la variable `img2`.
Enfin, l'instruction `img2.show()` vous permet à tout moment d'afficher l'image correspondante à la variable `img2`.

Partie I - Squelettisation « naïve » de l'image de l'empreinte digitale

Afin de réduire l'épaisseur de chaque ligne noire, nous allons procéder par un balayage de l'image de base en la parcourant de droite à gauche et de haut en bas. Lors de ce parcours, et à chaque étape, un groupe de deux pixels verticaux consécutifs sera examiné et éventuellement modifié selon le principe suivant : si ce bloc est constitué de deux pixels noirs, celui du haut deviendra blanc, sinon le bloc est inchangé.

Un exemple est donné sur un extrait d'image où quelques pixels sont représentés :

<table>
<tr><td colspan="2">Étape 1 : balayage simultané des lignes 1 et 2</td></tr>
<tr><td>Avant :</td><td>Après traitement des lignes :</td></tr>
<tr><td></td><td></td></tr>
<tr><td colspan="2">Étape 2 : balayage simultané des lignes 2 et 3</td></tr>
<tr><td>Avant :</td><td>Après traitement des lignes :</td></tr>
<tr><td></td><td></td></tr>
</table>

Et on continue ainsi de suite jusqu'au traitement des deux dernières lignes de l'image.

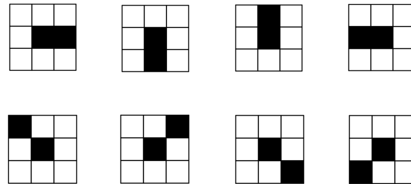
Travail à produire :

Concevoir un programme en Python qui utilise ce procédé pour créer le squelette d'une empreinte digitale. On utilisera à cet effet l'image contenue dans le fichier `Empreinte_digitale.png` et on enregistrera le résultat sous le nom `squelette.png` .
Le programme conçu est également à transmettre.

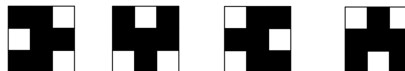
Partie II – Identification de certaines minuties

De nombreuses minuties peuvent être identifiées à partir d'un squelette reflétant précisément une empreinte digitale. Aussi nous contenterons-nous ici de l'identification de deux types de seulement : les *terminaisons* et les *bifurcations*.
Pour les définir, nous examinerons les pixels qui l'entourent.
Aussi :

- un pixel noir sera associé à une terminaison, si parmi les pixels qui sont dans son voisinage immédiat, un seul pixel est noir :



- un pixel noir sera associé à une bifurcation s'il se trouve au centre de l'une des configurations suivantes :



Travail à produire :

Concevoir un programme en Python qui marque de deux couleurs distinctes les minuties identifiées à partir du fichier `Skeleton.png`. On enregistrera l'image obtenue sous le nom `minuties.png` . Ce programme est également à transmettre.

Résoudre un défi du jeu *Par Odin*

1. Introduction

Le jeu *Par Odin* est une sorte de casse-tête en solitaire d'Antonin Boccara, publié par les éditions Oldchap. Dans sa version de base, le principe de ce jeu est le suivant :

- on dispose de 7 dés blancs identiques à 6 faces, avec pour chacun les 6 faces suivantes :
  - héros
  - capitaine
  - soldat
  - maudit
  - traître
  - mage
- un «défi» correspond à un tirage (aléatoire ou imposé) des 7 dés. Une fois le défi défini, on ne doit plus modifier les dés. Résoudre un défi consiste alors « simplement » à séparer les 7 dés en deux « armées » (2 groupes) de force identique.
- La force d'une armée est calculée ainsi :
  - chaque capitaine vaut 2 points de force
  - chaque soldat vaut 1 point de force
  - chaque maudit vaut -1 point de force
  - chaque héros vaut – normalement – 3 points de force
  - chaque traître vaut 1 point de force et annule la force d'**un et un seul** héros
  - chaque mage vaut 1 point de force par dé blanc de l'armée qui n'est pas un mage.

Exemple :

- On doit résoudre le défi suivant :

```
defi = [HEROS, HEROS, CAPITAINE, MAUDIT, TRAITRE, SOLDAT, MAGE]
```

- Une solution est alors la suivante :

```
armee1 = [HEROS, CAPITAINE, SOLDAT, TRAITRE]

armee2 = [MAUDIT, HEROS, MAGE]
```

En effet, calculons la force de chacune des 2 armées :

- armée 1 :
    - la force du HEROS est annulée par la présence du traître → 0
    - le CAPITAINE a une force de 2
    - le SOLDAT a une force de 1
    - le TRAITRE a une force de 1
    - → 0 + 2 + 1 + 1 = 4
- armée 2 :
    - le MAUDIT a une force de - 1
    - le HEROS a une force de 3
    - le MAGE a une force de 2
    - → - 1 + 3 + 2 = 4
- Les deux armées ont bien la même force.

Vous devrez écrire un programme capable de résoudre un défi proposé, mais aussi capable de proposer un défi tiré aléatoirement mais ayant au moins une solution. Le détail de ce qui est demandé figure au paragraphe suivant.

2. Travail à réaliser

2.1. Phase 1 : résoudre un défi

- Vous devez partir du programme `parodin_eleve.py` qui vous est fourni. Ce fichier contient déjà quelques éléments :
- des constantes utiles pour représenter les différentes faces de dés ;
- une fonction `verification_evaluer` qui vous permettra de vérifier votre fonction de calcul de la force d'une armée ;
- une fonction `verification_resoudre` qui vous permettra de vérifier que votre programme de résolution de défi fonctionne bien ;
- une fonction `resoudre` qui prend en paramètre un défi, et renvoie une solution à ce défi s'il en existe une. Sinon, elle renvoie `None`.
- Vous devez implanter les fonctions suivantes :
    - une fonction `lancer_des` qui génère un défi, que celui-ci ait une solution ou non ;
    - une fonction `evaluer`, qui calcule la force d'une armée ;

- une fonction `verification_decomposition_valide`, qui prend en paramètre un couple d'armées, et renvoie `True` si les deux armées ont la même force, `False` sinon. Cette fonction est appelée par la fonction `resoudre`.
- une fonction `comparer_couples`, qui prend en paramètre deux couples de listes et vérifie que ces deux couples sont égaux si l'on ne tient pas compte de l'ordre des composantes du couples et de l'ordre des éléments dans les listes. Cette fonction est utilisée par la fonction `verification_resoudre`. Cette fonction sert dans les cas de test pour vérifier que votre code est correct.

2.2. Phase 2 : générer un défi

Vous devez implanter une fonction `generer_defi`, qui renvoie un défi tiré aléatoirement, mais ayant au moins une solution.

2.3. Phase 3 : traiter un dé noir

Le jeu *Par Odin* fournit également un dé noir, présentant les faces suivantes :

- loup
- serpent
- cheval
- dragon
- sanglier
- aigle

- La force d'une armée est calculée ainsi :
  - Le loup Fenrir : multiplie par 2 le dé blanc le plus fort de son armée ;
  - Le serpent Jörmungand : inverse la force du dé blanc le plus fort de son armée ;
  - Le cheval Sleipnir : augmente de 1 chaque dé blanc de son armée ;
  - Le dragon Fáfnir : réduit de 1 chaque dé blanc de son armée ;
  - Le sanglier Gullinbursti : agit sur les deux armées, augmente de 1 chaque dé blanc identique d'une même armée ;
  - L'aigle Hraesvelg : agit sur les deux armées, réduit de 1 chaque dé blanc identique d'une même armée
- Ainsi, on peut avoir des défis plus compliqués à résoudre en ajoutant un dé noir au 7 dés blancs. Le fichier `complement.py` vous donne un ensemble de lignes à rajouter à votre fichier pour traiter les défis avec un dé noir. Modifiez votre programme pour qu'il puisse traiter ces défis. Traiter les différentes faces une à une. Le fichier `complement.py` comporte des fonctions pour tester ce type de défi.

3. Consignes générales

- Vos fonctions doivent être documentées avec une docstring.
- Vos fonctions et variables doivent avoir des noms explicites.
- Vos fonctions doivent être les plus courtes possibles (moins de 20 lignes, mais moins de 10 lignes est encore mieux). Pour y arriver, décomposez vos fonctions en sous-fonctions avec des noms explicites.
- Un programme qui marche mais qui ne fait pas tout est toujours préférable à un programme qui plante.