BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2023

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.
Ce sujet comporte 13 pages numérotées de 1/13 à 13/13.

Le candidat traite les 3 exercices proposés

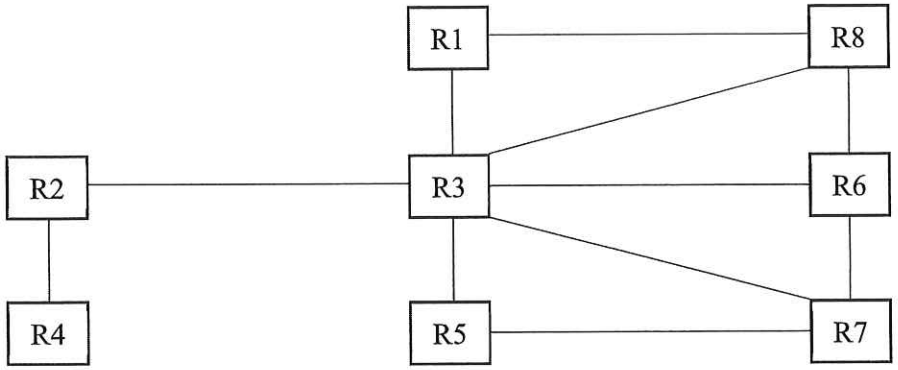
EXERCICE 1 (5 points)

Les deux parties sont indépendantes.

Partie A

La responsable informatique doit gérer le réseau informatique de son entreprise représenté ci-dessous dans lequel R1, R2, R3, R4, R5, R6, R7 et R8 sont des routeurs. Elle décide d'utiliser le protocole RIP pour configurer les tables de routages.

Ce protocole est un protocole à vecteurs de distance. La métrique permettant de décider du meilleur chemin vers un routeur distant est le nombre de sauts.



1. La table de routage de R1 débute ainsi :

Destination	Passerelle	Métrique
R1	R1	0
R2	R3	2
R3	R3	1

Recopier puis compléter la table de routage de R1 selon le protocole RIP.

On donne le constructeur de la classe `Routeur`

```
class Routeur :
    def __init__(self, name) :
        self.nom = name
        self.table_routage = { self : (self, 0) }
        # le routeur est lié à lui-même (self)
```

L'attribut `table_routage` est un dictionnaire dont les clés sont des objets de type `Routeur` et la valeur est le couple `(pasr, m)` où `pasr` est un objet de type `Routeur` et `m` est la métrique selon le protocole RIP entre le routeur de la clé et le routeur `pasr`.

2. La méthode `ajout_destination(self, dest, pasr, m)` de la classe `Routeur`, prend en paramètres un routeur de destination `dest`, un routeur passerelle `pasr` et un entier `m`. Elle ajoute à la table de routage de `self` le routeur `dest` associé à la passerelle `pasr` et à la métrique `m`.

   Par exemple, on crée les routeurs R1, R2 et R3 puis on ajoute les destinations R2 et R3 à la table de routage de R1 à l'aide des commandes suivantes :

```
r1 = Routeur("R1")
r2 = Routeur("R2")
r3 = Routeur("R3")
r1.ajout_destination(r2, r3, 2)
r1.ajout_destination(r3, r3, 1)
```

   On dispose d'une méthode `afficher(self)` qui permet d'obtenir la table de routage du routeur `self`.

   Par exemple, la commande `r1.afficher()` affiche :

```
      ROUTEUR R1 : table de routage
dest.     | pass.    | metrique
R1        | R1       | 0
R2        | R3       | 2
R3        | R3       | 1
```

   a. Proposer les commandes nécessaires permettant d'ajouter les routeurs R4, R5 à la table de routage de R1.

   b. Écrire la méthode `ajout_destination(self, dest, pasr, m)`.

3. Écrire la méthode `voisins(self)` qui renvoie la liste des routeurs directement connectés au routeur `self`.

   Par exemple, `r1.voisins()` renverra la liste `[r3, r8]`, où `r3` et `r8` sont des objets de types `Routeur`.

4. La méthode `calcul_route(self, dest)` de la classe `Routeur` prend en paramètre un routeur de destination `dest` et renvoie la liste des routeurs parcourus lors d'une communication entre les routeurs `self` et `dest`.

Par exemple, `r1.calcul_route(r7)` renverra la liste `[r1, r3, r7]` où `r1`, `r3` et `r7` sont des objets de types `Routeur`.

Recopier et compléter le code de la méthode `calcul_route(self, dest)` ci-dessous

```
def calcul_route(self, dest) :
    route = [self]
    routeur_courant = route[-1]
    while ... :

        ...          # plusieurs lignes

    return route
```

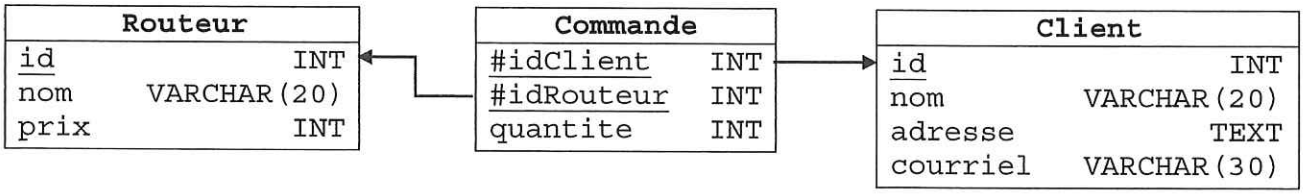
Partie B

Dans cette partie, on pourra utiliser les mots clés suivants du langage SQL.

`SELECT, INSERT INTO, WHERE, UPDATE, JOIN, ORDER BY`

La commande `ORDER BY propriété` permet de trier dans l'ordre croissant les résultats d'une requête selon l'attribut `propriété`.

Pour gérer son réseau informatique, la responsable achète son matériel à la société BeauReseau qui dispose d'une base de données dont le schéma relationnel est ci-dessous.



Les clés primaires sont soulignées et les clés étrangères sont précédées du caractère #.
Ainsi l'attribut `idRouteur` de la relation `Commande` est une clé étrangère qui fait référence à l'attribut `id` de la relation `Routeur` et l'attribut `idClient` de la relation `Commande` est une clé étrangère qui fait référence à l'attribut `id` de la relation `Client`.

Pour les questions où on demande des résultats de requêtes, on considèrera les extraits des tables remplies ainsi :

Table Routeur

id	nom	prix
1	C6Po-1000	1200
2	Coq6-300	6000
3	Al-200	6000
4	C6Po-9000	9000

Table Client

id	nom	adresse	courriel
1	Knuth	rue Donald, Tampa	dknuth@usa.org
2	Hooper	rue Grace, Boston	ghooper@usa.org
3	Torvalds	rue Linus, Helsinki	ltorvalds@finland.org
4	Pouzin	rue Louis, Paris	lpouzin@france.fr

Table Commande

idClient	idRouteur	quantite
1	1	4
1	3	1
2	1	1
2	1	5
3	2	3
4	4	1
4	1	5

5. On considère la requête d'insertion suivante

```
INSERT INTO Routeur(id, nom, prix) VALUES(3, 'Dali-32', 4000)
```

Expliquer pourquoi cette requête renvoie une erreur.

6. Proposer une requête qui renvoie le nom de tous les routeurs dont le prix est compris entre 2500 (inclus) et 7000 euro (inclus).

7. On considère la requête suivante :

```
SELECT nom FROM Client
JOIN Commande ON Commande.idClient = Client.id
WHERE Commande.quantite = 1
```

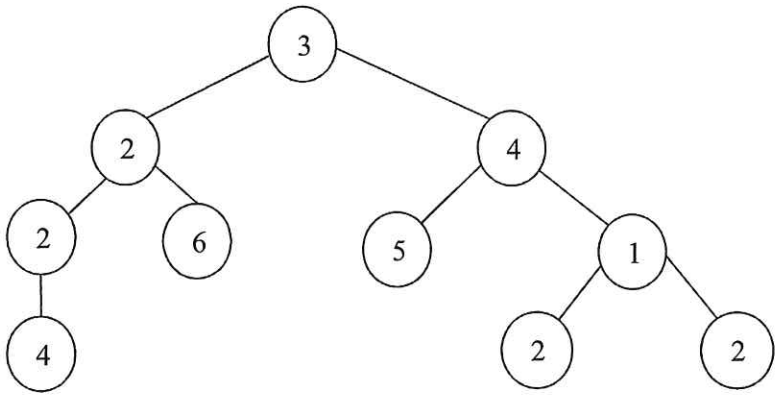
En considérant les extraits des tables fournies, préciser ce que renvoie cette requête.

8. Proposer une requête qui renvoie les noms triés dans l'ordre croissant des routeurs achetés par le client n°2.

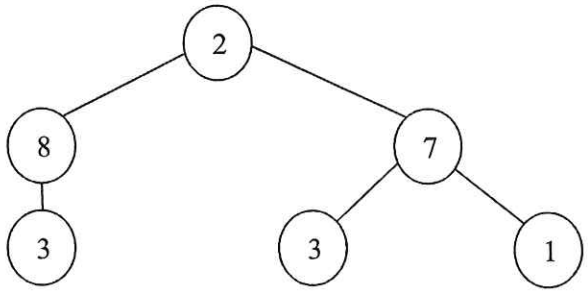
EXERCICE 2 (3 points)

Tous les arbres de cet exercice auront des nœuds avec au plus deux enfants.
On considère des arbres dont les étiquettes sont des nombres entiers positifs.
On appelle poids d'un tel arbre la somme de toutes ses étiquettes.
On dira qu'un arbre est un arbre mobile si pour chaque nœud, tous ses sous-arbres sont des arbres mobiles de même poids.

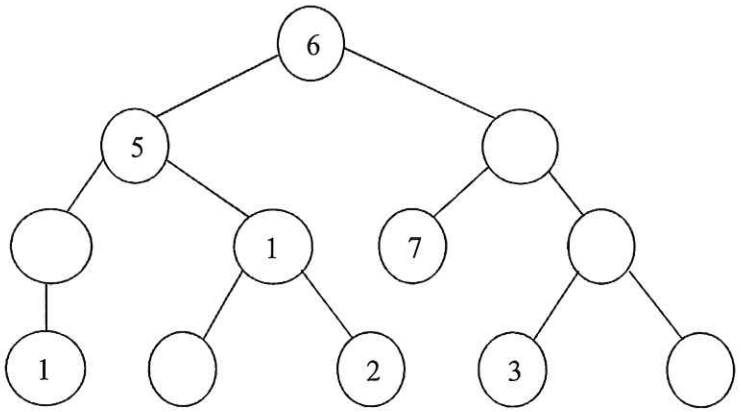
Voici un exemple d'arbre mobile.



Voici un exemple d'arbre non-mobile.



1. Recopier et compléter l'arbre ci-dessous de sorte qu'il soit un arbre mobile.



2. Chaque nœud d'un arbre sera représenté en Python par une liste `[e, lst_sa]` où :
   - `e` est l'étiquette du nœud ;
   - `lst_sa` est la liste des sous-arbres de ce nœud.

   L'arbre vide sera représenté par une liste vide.
   Un arbre est ainsi représenté par une liste correspondant à son nœud racine.

   On considère le script Python ci-dessous.

```
n2 = [2, []]
n5 = [5, []]
n8 = [8, []]
n1 = [1, [n2, n5]]
a = [4, [n1, n8]]
```

   a. Dessiner sur votre copie l'arbre représenté par la variable `a`.

   b. Cet arbre n'est pas un arbre mobile, mais on peut le transformer en arbre mobile en ajoutant exactement deux nœuds. Dessiner l'arbre mobile ainsi obtenu et écrire des instructions à ajouter au script précédent pour que la variable `a` représente cet arbre mobile.

3. Le poids d'un arbre peut se calculer de la manière suivante : on parcourt les nœuds de l'arbre en largeur à l'aide d'une file en maintenant à jour une variable `p` égale à la somme des étiquettes des nœuds parcourus.

   On utilisera l'interface des files suivante :

`creer_file()`	Renvoie une file vide
`est_vide(f)`	Renvoie `True` si la file `f` est vide, `False` sinon
`enfiler(f, e)`	Ajoute l'élément `e` dans la file `f`
`defiler(f)`	Supprime et renvoie l'élément en tête de la file `f`

   La fonction `poids(arbre)` a pour paramètre un arbre `arbre` et renvoie son poids. Recopier et compléter les lignes 7, 8 et 11 du code ci-dessous.

```
1.  def poids(arbre):
2.      if arbre == [] :
3.          return 0
4.      p = 0
5.      file = creer_file()
6.      enfiler(file, arbre)
7.      while ... :
8.          arbre = ...
9.          for sous_arbre in arbre[1] :
10.             enfiler(file, sous_arbre)
11.         p = ...
12.     return p
```

4. Pour déterminer récursivement si un arbre est un arbre mobile, on doit :

- vérifier que tous ses sous-arbres ont le même poids ;
- vérifier récursivement que tous ses sous-arbres sont des arbres mobiles.

Écrire une fonction récursive `est_mobile(arbre)` qui a pour paramètre un arbre `arbre` et renvoie `True` si cet arbre est un arbre mobile, et `False` sinon.
Pour cela, on pourra utiliser la fonction `poids`.
On rappelle que l'arbre vide est un arbre mobile.

EXERCICE 3 (4 points)

En informatique, chaque pixel d'une image numérique est affiché sur un écran standard par synthèse additive du rouge, du vert et du bleu : c'est le système colorimétrique RVB. Chacune des trois composantes RVB d'un pixel est stockée sur un octet.

Pour représenter une image, un fichier au format *Windows bitmap* (BMP) contient un entête, suivi des valeurs des composantes RVB de chacun des pixels, écrites les unes à la suite des autres.

1. On considère une image de hauteur 1000 pixels et de largeur 1500 pixels.
   Combien de mégaoctets faut-il pour représenter l'ensemble des pixels de cette image en omettant l'entête ?

2. En Python, les images sont généralement représentées à l'aide de tableaux de tableaux.
   - les composantes RVB d'un pixel sont stockées dans un `tuple` de trois entiers compris entre 0 et 255 ;
   - pour chaque ligne de l'image, les composantes RVB des pixels sont stockées dans un tableau ;
   - l'image est alors représentée par un tableau `img` contenant tous les tableaux précédents ; par convention, l'image vide est représentée par le tableau contenant un tableau vide `[[]]`.

   Ainsi, si l'image `img` a une hauteur de 1000 pixels et largeur de 1500 pixels,

```
>>> len(img)
1000
>>> len(img[0])
1500
```

   La position de chaque pixel dans une image `img` est repérée par un couple d'entiers `(i, j)` qui sont respectivement les indices de ligne et de colonne du pixel dans `img`.

   Encadrer `i` et `j` lorsque l'image `img` a pour hauteur `n` et largeur `m`.

Lorsque `i` et `j` vérifient ces encadrements, on dira qu'ils sont **compatibles** avec la taille de l'image.

On s'intéresse dans la suite de cet exercice à l'algorithme de recadrage intelligent (*seam carving*), qui permet de réduire la largeur d'une image tout en conservant ses principales caractéristiques graphiques.

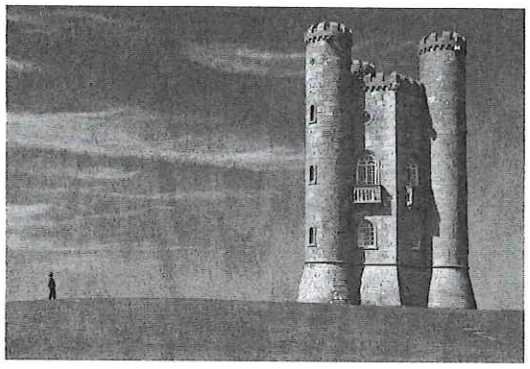
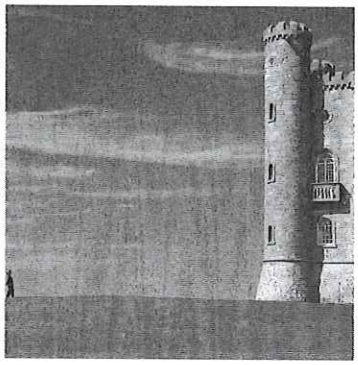
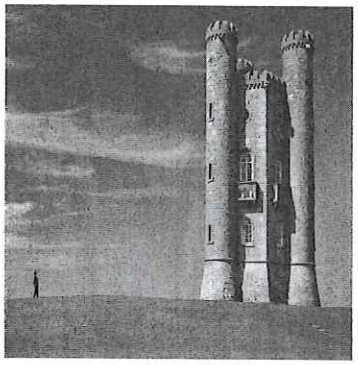


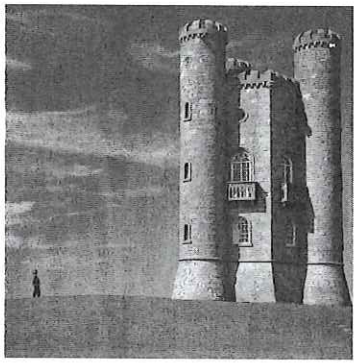
Image dont on souhaite réduire la largeur



Découpe de l'image
une partie du château n'est plus visible



Redimensionnement
le château est déformé

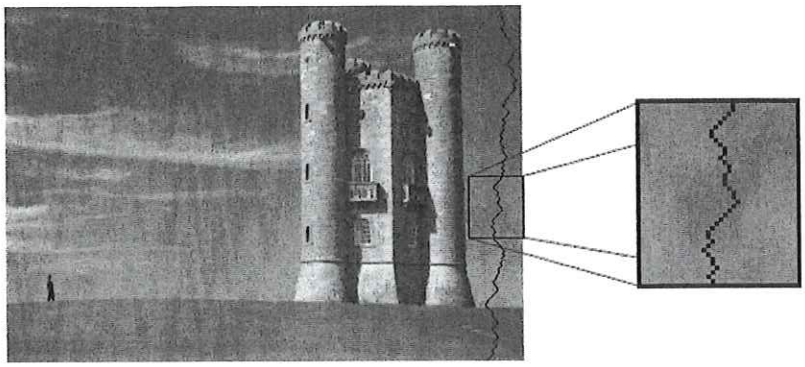


Seam carving
les aspects des principaux éléments graphiques sont conservés

Source des illustrations : https://en.wikipedia.org/wiki/Seam_carving

L'algorithme de recadrage intelligent s'appuie sur la suppression de pixels « bien choisis » dans l'image. Pour déterminer ces pixels, il recherche ce que l'on appellera la « couture de moindre énergie ».

Une couture est une suite de pixels adjacents allant du haut au bas de l'image. On associe à chaque pixel de l'image un nombre positif appelé énergie du pixel. L'énergie d'une couture est alors la somme des énergies des pixels qui la composent.



Une couture et un détail de cette couture.

L'algorithme de recadrage intelligent d'une image de dimensions `n` × `m` (hauteur `n`, largeur `m`) détermine alors la couture de plus basse énergie et supprime de l'image tous les pixels présents dans la couture. L'image résultante est de dimensions `n` × `(m - 1)` (hauteur `n`, largeur `m - 1`). On recommence le procédé jusqu'à obtenir la largeur souhaitée.

3. On suppose que l'on dispose d'une fonction `energie(img, i, j)`, qui prend en paramètres une image `img`, un indice de ligne `i`, un indice de colonne `j` et renvoie l'énergie du pixel à la position `(i, j)` de l'image.
On décide de calculer au préalable les énergies des différents pixels de l'image. La fonction `calcule_tab_energie` prend comme paramètre une image `img` et renvoie le tableau de tableaux des énergies correspondantes. Ainsi `tab_energie[i][j]` contiendra à la fin de l'exécution l'énergie du pixel à la position `(i, j)`.

```
def calcule_tab_energie(img):
    n = len(img)
    m = len(img[0])
    tab_energie = []
    for i in range(n):
        ligne = []
        for j in range(m):
            e = energie(img, i, j)
            ligne.append(e)
        tab_energie.append(ligne)
    return tab_energie
```

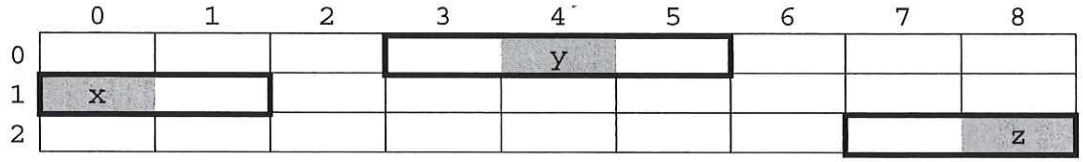
a. Quel est le type de la variable `tab_energie` renvoyée par la fonction `calcule_tab_energie(img)` ?

b. Exprimer en fonction de `n` et de `m` le nombre d'appels à la fonction `energie`.

4. On représente en Python une couture à l'aide d'un tableau de couples d'indices `(i, j)` compatibles avec la taille de l'image. Chaque couple `(i, j)` repère la position dans l'image d'un pixel de la couture.
On rappelle que l'énergie d'une couture est la somme des énergies des pixels qui la composent.

Écrire une fonction `calcule_energie(couture, tab_energie)`, qui prend en paramètres un tableau `couture` représentant une couture ainsi que le tableau des énergies `tab_energie` renvoyé par la fonction `calcule_tab_energie`. Cette fonction doit renvoyer l'énergie de la couture passée en argument.

5. Écrire la fonction `indices_proches(m, i, j)`, qui prend en paramètres `m` le nombre de colonnes de l'image, `i` un indice de ligne, `j` un indice de colonne, et renvoie parmi les positions des pixels `(i,j-1)`, `(i,j)` et `(i,j+1)` celles qui sont compatibles avec les dimensions de l'image.
On ne tiendra pas compte de l'ordre des éléments de la liste renvoyée.
Par exemple, on a repéré dans l'image de dimension 3×9 ci-dessous trois pixels `x`, `y`, `z`.




```
>>> indices_proches(9, 1, 0) # pixels voisins de x
[(1,0), (1,1)]
>>> indices_proches(9, 0, 4) # pixels voisins de y
[(0,3), (0,4), (0,5)]
>>> indices_proches(9, 2, 8) # pixels voisins de z
[(2,7), (2,8)]
```

6. On appellera couture d'indice `j` une couture dont la position du premier pixel est `(0, j)`. Afin de déterminer une couture de basse énergie parmi toutes les coutures d'indice `j`, on met en œuvre une stratégie de type glouton.
Pour cela, on construit la couture du haut vers le bas en sélectionnant à chaque étape le pixel d'énergie minimale parmi ceux situés immédiatement en bas à gauche, immédiatement en bas ou immédiatement en bas à droite lorsque cela est possible.

Par exemple, on donne ci-dessous un tableau `tab_energie` d'une image de dimension `4 × 3`. Les pixels de la couture d'indice `j = 1` y ont été grisés.

	0	1	2
0	3	4	2
1	4	5	6
2	3	2	1
3	7	8	6

Le premier pixel de la couture a pour position `(0, 1)`
Le 2e pixel sera choisi parmi `[(1,0), (1,1), (1,2)]`
Le 3e pixel sera choisi parmi `[(2,0), (2,1)]`
Le 4e pixel sera choisi parmi `[(3,0), (3,1), (3,2)]`

La couture d'indice 1 est donc : `[(0,1), (1, 0), (2, 1), (3, 2)]`

La fonction `calcule_couture(j, tab_energie)`, ci-dessous a pour paramètres un entier `j` et le tableau des énergies `tab_energie`. Cette fonction doit renvoyer une liste de tuples représentant la couture d'indice `j` construite selon le procédé décrit précédemment.
On pourra utiliser la fonction `indices_proches`, ainsi que la fonction `min_energie(tab_energie, indices_pixels)`, qui prend en paramètres le tableau d'énergie `tab_energie` et la liste de positions de pixels `indices_pixels`. La fonction `min_energie` renvoie le tuple `(i,j)` de la position du pixel de moindre énergie.

Recopier et compléter le code de la fonction `calcule_couture` ci-dessous :

```
def calcule_couture(j, tab_energie) :
    n = len(tab_energie)      # hauteur de l'image
    m = len(tab_energie[0]) # largeur de l'image
    couture = []
    couture.append((0, j))    # premier pixel de la couture
    for i in range(1, n):     # i est l'indice de la ligne du
                              # prochain pixel dans la couture
        ...                   # plusieurs lignes possibles

    return couture
```

7. Le paramètre `j` de la fonction précédente peut prendre toutes les valeurs entières comprises entre `0` (inclus) et `m = len(tab_energie[0])` (exclu).
Écrire une fonction `meilleure_couture(tab_energie)`, qui renvoie une couture d'énergie minimale parmi celles obtenues à l'aide de la fonction `calcule_couture`.
L'algorithme consistera à parcourir tous les indices `j` et ne garde qu'une couture d'indice `j` qui soit d'énergie minimale. Pour cela, on pourra utiliser les fonctions `calcule_couture` et `calcule_energie` définies auparavant.