Bac Général Maths + NSI (Mathématiques et Numérique et sciences informatiques)

SUJET D'EXAMEN (SESSION OFFICIELLE)

Numérique et sciences informatiques - Session 2025

Sujet de l'examen national tel que publié pour la session indiquée. Source : sujets nationaux publiés par le ministère chargé de l'examen et les services académiques. Document mis à disposition par FormaV pour la révision.

Toutes les annales et leurs corrigés : https://bac-maths-nsi.fr/annales-sujets-et-corriges

Amérique du nord - mai 2025 - sujet 3

Exercice 1 (Programmation python et programmation dynamique - 6 points)
Dans cet exercice, on se réfère à la citation suivante de Donald Knuth :

« *An algorithm must be seen to be believed.* »

En typographie, il existe plusieurs manières d'aligner un texte : aligner à gauche, centrer, aligner à droite et justifier.

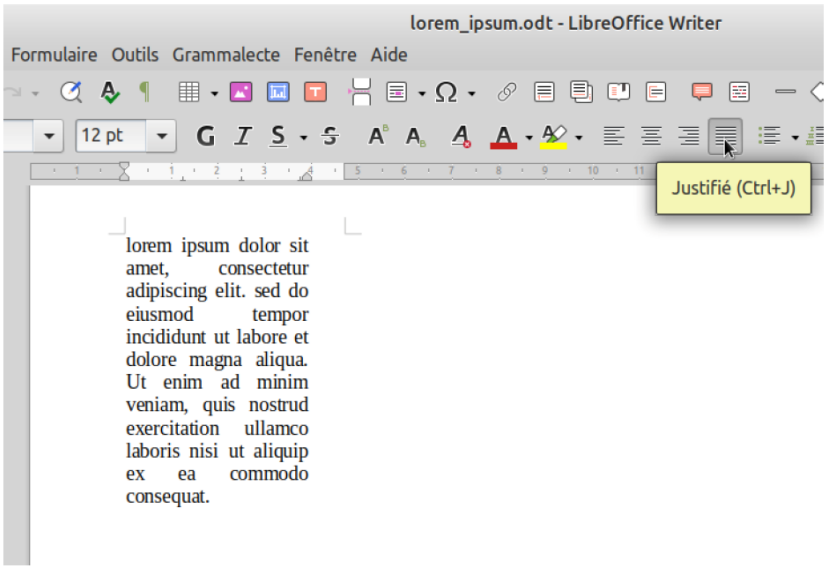


FIGURE 1 – Justification d'un texte à l'aide d'un traitement de texte

L'alignement justifié permet d'obtenir que chaque ligne ait la même longueur appelée *justification*. Pour cela et pour chacune des lignes, on ajoute si nécessaire des espaces supplémentaires. Dans tout cet exercice, on désigne par *un espace* exactement un caractère d'espacement, tel que contenu dans la chaine Python `' '`.

On répartit ces espaces supplémentaires ainsi :

- ⋆ s'il n'y a qu'un mot on insère les espaces à droite de celui-ci ;
- ⋆ sinon :
  - on effectue la division euclidienne du nombre total d'espaces nécessaires pour compléter la ligne par le nombre d'emplacements inter-mots (entre les mots) ;
  - on répartit le quotient d'espaces entre chaque emplacement inter-mot ;
  - puis, s'il reste des espaces à distribuer, on les répartit une par une de gauche à droite.

Dans toute la suite, pour simplifier, on considère que tous les caractères, espaces comprises, ont la même largeur. Les mots ne sont ni coupés, ni décorés (gras, italique, etc.).

Partie A

1. On considère les quatre premiers mots `'An'`, `'algorithm'`, `'must'`, `'be'` de la citation de Donald Knuth avec un alignement justifié de 25 caractères, c'est-à-dire que la ligne contient 25 caractères au total, en comptant les lettres et les espaces. Montrer que, pour cette justification, le nombre d'espaces nécessaires est de 8.
2. On souhaite répartir ces huit espaces entre ces quatre mots. Déterminer la seule proposition, parmi les quatre propositions ci-après, respectant les règles de l'alignement justifié définies précédemment, pour une justification de 25 caractères. Le caractère - représente ici une espace.

   ⋆ `An--algorithm---must---be`
   ⋆ `An---algorithm---must--be`
   ⋆ `An----algorithm--must--be`
   ⋆ `An---algorithm--must---be`

On considère le code de la fonction `ajout_espace`, donnée ci-après, qui prend en paramètres une liste `liste_mots` non vide de chaînes de caractères représentant un ensemble de mots et un entier `justification` représentant la justification. La fonction renvoie une chaîne de caractères constituée des mots de `liste_mots` à laquelle on ajoute des espaces pour la justifier selon `justification`.

La fonction `sum` en Python est utilisée pour calculer la somme des éléments contenus dans une liste.

On rappelle qu'en Python, le caractère \ placé en fin de ligne permet de continuer l'expression Python sur la ligne suivante, comme si on n'avait en fait pas sauté de ligne. Cela sert uniquement à améliorer la présentation du code en évitant d'avoir des lignes trop longues. Par exemple les deux lignes :

```
reponse = reponse + " " * ... \
+ liste_mots[i]
```

signifient exactement la même chose que la ligne :

```
reponse = reponse + " " * ... + liste_mots[i]
```

```
1  def ajout_espace(liste_mots: list[str],
2                   justification: int) -> str:
3      nb_caracteres = sum([len(mot) for mot in liste_mots])
4      nb_mots = len(liste_mots)
5      assert nb_caracteres + ...
6      nb_espace_total = justification - nb_caracteres
7      if nb_mots == 1:
8          return ... + " " * nb_espace_total
9      else:
10         q = nb_espace_total // (nb_mots - 1)
11         r = nb_espace_total % (nb_mots - 1)
12         reponse = liste_mots[0]
13         for i in range(1, r + 1):
14             reponse = reponse + " " * ... \
15                 + liste_mots[i]
16         for i in range(r + 1, nb_mots):
17             reponse = reponse + " " * ... \
18                 + liste_mots[i]
19     return reponse
```

3. Dans la fonction `ajout_espace` ci-dessus, il peut arriver que la liste de mots `liste_mots` soit trop longue pour tenir sur une seule ligne de justification donnée. Compléter la précondition de cette fonction à la ligne 5 avec un critère que doit vérifier la liste `liste_mots`, pour être justifiable sur une seule ligne.

Dans le cas d'une liste d'au moins deux mots, si on a `nb_espace_total` espaces à répartir entre `nb_mots - 1` emplacements inter-mots selon les règles de l'alignement justifié, avec `q` et `r` respectivement le quotient et le reste de la division euclidienne de `nb_espace_total` par `nb_mots - 1`, il suffit de :

- placer `q+1` espaces pour les `r` premiers emplacements inter-mots ;
- placer `q` espaces pour les emplacements inter-mots suivants allant de `r+1` à `nb_mots - 1`.

4. Recopier et compléter les lignes 8, 14 et 17 du code de la fonction `ajout_espace`.

Partie B

Dans cette partie on cherche à déterminer après quel mot revenir à la ligne. On admet que la justification est supérieure à la longueur des mots considérés, ainsi chaque mot peut tenir sur une ligne.

5. Proposer un algorithme en langage naturel permettant de déterminer après quel mot d'un texte revenir à la ligne, étant donné une certaine justification. On attend uniquement une explication précise de l'algorithme, pas sa programmation en Python.

On utilise une liste de tuples pour modéliser le découpage d'un texte en différentes lignes. Par exemple, pour la liste de mots

```
['An', 'algorithm', 'must', 'be', 'seen', 'to', 'be', 'believed'],
```

la liste de découpage `[(0, 2), (2, 5), (5, 7), (7, 8)]` signifie que :

- la première ligne sera constituée des mots d'indice de 0 à 2 (2 exclu), soit `An algorithm` ;
- la seconde ligne des mots de 2 à 5 (5 exclu), soit `must be seen` ;
- la troisième ligne des mots de 5 à 7 (7 exclu), soit `to be` ;
- et la dernière ligne des mots de 7 à 8 (8 exclu), soit `believed`.

6. Recopier et compléter les lignes 5 à 8 du code de la fonction `affiche_justifie`, donné ci-après, qui prend en paramètres une liste de mots, une liste de découpage et une justification puis affiche dans la console Python les lignes justifiées correspondantes.
Remarque : `liste_mots[a:b]` est une tranche (ou *slice* en anglais). Elle désigne la liste extraite de la liste `liste_mots` constituée des éléments dont l'indice est compris entre `a` et `b-1` inclus.

```
1 def affiche_justifie(liste_mots: list[str],
2                       decoupage: list[(int, int)],
3                       justification: int) -> None:
4     # début de la boucle d'affichage justifié
5     for ... in decoupage:
6         ligne_justifiee = \
7             ajout_espace(liste_mots[ ... : ... ], ...)
8         ...
```

Partie C

À l'usage, on se rend compte que la méthode précédente ne produit pas toujours un alignement justifié « esthétique ». Pour remédier à ce problème, les typographes utilisent une formule mathématique qui mesure la qualité esthétique d'une ligne en fonction du nombre d'espaces supplémentaires ajoutées. Entre deux mots d'une même ligne il y a forcément un espace. Sont donc considérées comme espaces supplémentaires ceux que l'on doit ajouter en plus.

Dans cette partie, on utilise une fonction qui mesure le défaut d'esthétique, appelé coût inesthétique, que l'on cherche à minimiser.

- Le coût inesthétique d'une ligne est le carré du nombre d'espaces supplémentaires nécessaires à la justification de cette ligne.
- Le coût inesthétique d'un texte pour un découpage donné est la somme du coût inesthétique de chaque ligne.

L'objectif de cette partie est, pour un texte donné, de proposer un découpage minimisant ce coût.

7. Étant donné une justification de 15 caractères, la liste de mots `['An', 'algorithm', 'must', 'be', 'seen', 'to', 'b`
et la liste de découpage `[(0, 2), (2, 4), (4, 7), (7, 8)]`, reproduire et compléter les trois dernières lignes du tableau ci-après (chaque ligne du tableau correspond à une ligne du texte découpé).

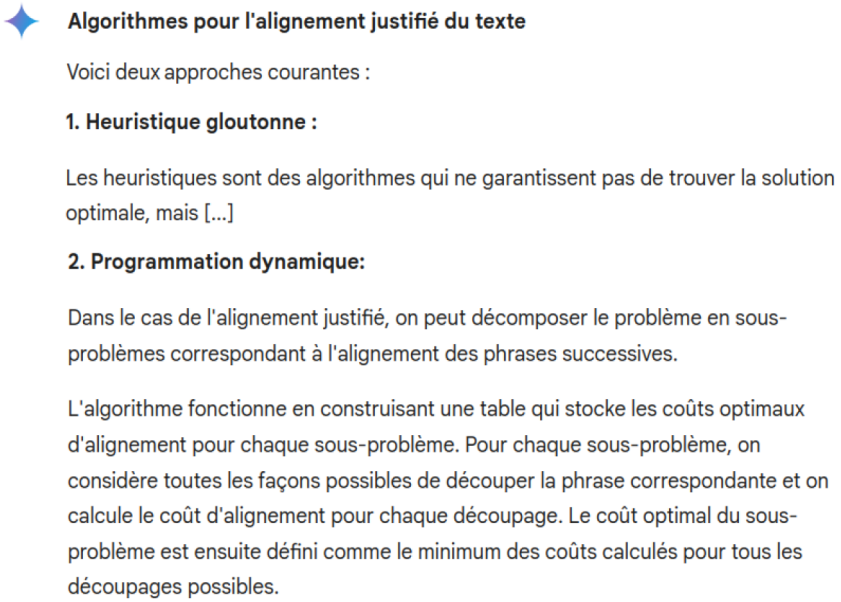
Coût total du découpage : 147					
Indice du mot de début	Indice du mot de fin +1	Nombre de mots	Nombre de caractères	Nombre d'espaces supplémentaires pour atteindre 15 caractères	coût
0	2	2	11	3	9
2	4				
4	7				
7	8				

8. Ecrire le code d'une fonction `cout` qui prend en paramètres deux indices `i` et `j`, une liste de chaînes de caractères `liste_mots` et un entier `justification` représentant le nombre total de caractères souhaités par ligne. Elle renvoie le coût inesthétique de la ligne commençant au mot `liste_mots[i]` et finissant au mot `liste_mots[j-1]` si le nombre de caractères (espaces inter-mots comprises) de l'ensemble de ces mots est inférieur ou égal à `justification` et un million sinon.
Exemples :

```
>>> liste_mots = ['An', 'algorithm', 'must', 'be', 'seen', 'to', 'be', 'believed']
>>> cout(0, 2, liste_mots, 15)
9
>>> cout(0, 4, liste_mots, 15)
1000000
>>> cout(5, 8, liste_mots, 15)
1
```

9. Étant donnée une liste de `n` mots (avec `n`$\geqslant 50$), en remarquant qu'on peut revenir à la ligne ou non après chaque mot sauf le dernier, indiquer s'il est raisonnable de déterminer une solution minimisant le coût inesthétique en testant toutes ces possibilités.

On pose la question à une IA générative, qui propose deux méthodes dont voici un extrait dans la capture d'écran donnée ci-après.



Algorithmes pour l'alignement justifié du texte

Voici deux approches courantes :

1. Heuristique gloutonne :

Les heuristiques sont des algorithmes qui ne garantissent pas de trouver la solution optimale, mais [...]

2. Programmation dynamique:

Dans le cas de l'alignement justifié, on peut décomposer le problème en sous-problèmes correspondant à l'alignement des phrases successives.

L'algorithme fonctionne en construisant une table qui stocke les coûts optimaux d'alignement pour chaque sous-problème. Pour chaque sous-problème, on considère toutes les façons possibles de découper la phrase correspondante et on calcule le coût d'alignement pour chaque découpage. Le coût optimal du sous-problème est ensuite défini comme le minimum des coûts calculés pour tous les découpages possibles.

La première méthode consiste en un algorithme glouton, la seconde utilise la programmation dynamique.

On demande alors à l'IA de générer un code en Python d'une fonction `justifie_dynamique` qui prend en paramètres une liste de chaînes de caractères `liste_mots` et un entier `justification` représentant le nombre total de caractères souhaités par ligne, en utilisant la programmation dynamique. L'IA propose alors le code ci-après.

```
def justifie_dynamique(liste_mots: list[str],
                       justification: int)->list[(int,
                                                  int)]:
    """Renvoie une liste contenant un découpage de
       'liste_mots' justifiée selon 'justification'."""
    n = len(liste_mots)
    # génération de deux listes de n éléments
    cout_mini = [0] * n
    indice_retour_ligne_mini = [0] * n
    # parcours à rebours de la liste des mots
    for i in range(n-1, -1, -1):
        # initialisation du coût du découpage du ième
        # au dernier mot
        cout_mini[i] = cout(i, n, liste_mots,
                            justification)
        indice_mini = n
        # recherche d'un indice j minimisant le coût
        # total menant à la justification de i à j
        for j in range(i+1,n):
            best = cout_mini[j]\
                + cout(i, j, liste_mots, justification)
            if best < cout_mini[i]:
                cout_mini[i] = best
                indice_mini = j
        indice_retour_ligne_mini[i] = indice_mini
    # reconstruction de la liste decoupage en partant
    # du premier mot indice k=0
    decoupage = []
    k = 0
    while k < n:
        decoupage.append((k,
                          indice_retour_ligne_mini[k]))
        k = indice_retour_ligne_mini[k]
    return decoupage
```

10. Donner l'ordre de grandeur du nombre d'appels à la fonction `cout` lorsqu'on exécute la fonction `justifie_dynamique` en fonction de `n` le nombre de mots dans la liste.
11. Etablir à partir du code de la fonction `justifie_dynamique` la relation qui existe entre les éléments de la liste `cout_mini`.
12. Proposer une modification de la fonction `justifie_dynamique` pour qu'elle renvoie, en plus du découpage, le coût inesthétique de celui-ci.

Exercice 2 (Arbres binaires et représentation binaire - 6 points)

L'échange de données sur les réseaux devient de plus en plus important. Il s'avère que la compression est un élément essentiel dans cet échange. On étudie ici un algorithme de compression, reposant sur le codage de Huffman.

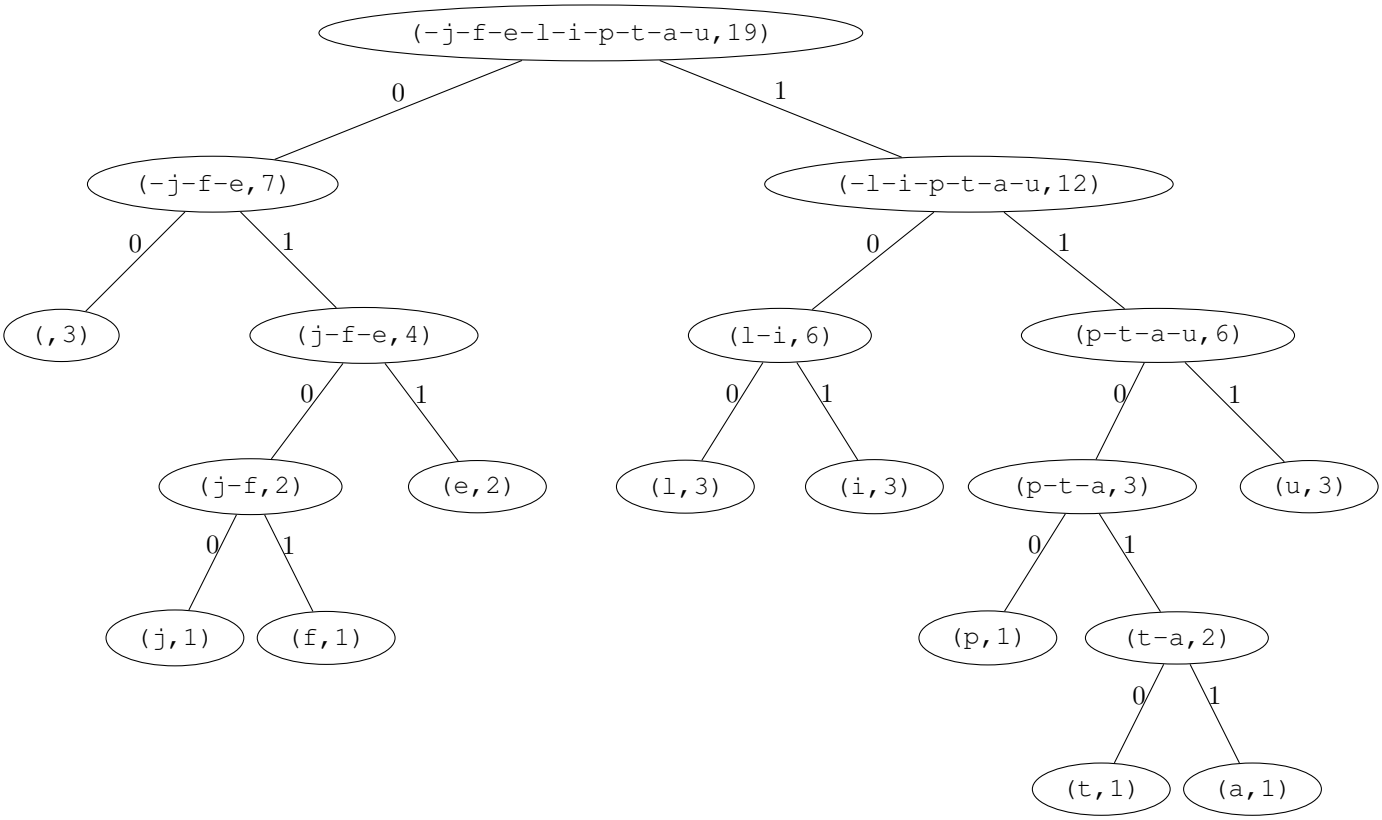


FIGURE 1 – `arb_julie`, un arbre de compression

Le codage de Huffman est adapté au texte à compresser car on se sert du nombre d'occurrences des caractères dans le texte. Par exemple, l'arbre de la Figure 1 a été construit pour compresser la phrase `"julie fuit la pluie"`. Le nombre associé à un ensemble de caractères correspond à la somme de leur nombre d'occurrences dans la phrase. Chaque chemin parcouru de haut en bas dans l'arbre définit le code associé au caractère présent en bas : ce code est constitué d'un 0 à chaque fois qu'on est allé à gauche et d'un 1 à chaque fois qu'fon est allé à droite, mis bout à bout.

La compression de `"julie"` donne donc : `j` → 0100 (gauche, droite, gauche, gauche), `u` → 111, `l` → 100, `i` → 101, `e` → 011, soit 0100111100101011. La compression de `" "` (caractère espace) donne 00.

1. Donner un exemple de feuille et la racine de l'arbre de la Figure 1.

On rappelle que la profondeur d'un nœud est égale à la distance en nombre d'arêtes entre ce nœud et la racine.

2. Donner la profondeur du nœud correspondant au caractère `p` et son code binaire associé.
3. On remarque que sur la Figure 1, les nœuds associés aux caractères les plus fréquents (avec un nombre d'occurrence plus élevé) ont une profondeur plus petite. Expliquer l'intérêt de cette propriété pour le codage binaire de la phrase.

On définit ci-dessous de façon incomplète la classe `Noeud`, où l'attribut `nom` est une chaîne de caractères représentant un ensemble de caractères séparés par des tirets (-) et où l'attribut `nb_occu` désigne la somme des nombres d'occurrences de ces caractères :

```
class Noeud:
    def __init__(self, nom, nb_occu, fils_g, fils_d):
        ....nom = nom
        ....nb_occu = nb_occu
        ....fils_g = fils_g
        ....fils_d = fils_d

    def __str__(self):
        """
        Renvoie une chaine contenant les donnees
        du noeud (nom et nombre d'occurrences)
        """
        return '(' + ... .nom + ',' + str(...) + ')'
```

4. Recopier et compléter les lignes 3, 4, 5, 6 et 13 de la classe `Noeud` ci-dessus.

On souhaite créer une fonction `liste_occurrences` qui à partir d'une chaîne de caractères `chaine` renvoie la liste des tuples `(c, nb_occu)` où `c` est un caractère apparaissant dans `chaine` et `nb_occu` est son nombre d'occurrences. Cette liste permettra par la suite de construire l'arbre qui codera chaque caractère comme en Figure 1.

Exemple :
```
>>> liste_occurrences('julie fuit la pluie')
[('j', 1), ('u', 3), ('l', 3), ('i', 3), ('e', 2),
(' ', 3), ('f', 1), ('t', 1), ('a', 1), ('p', 1)]
```

```
1  def liste_occurrences(chaine):
2      dico = ...
3      for c in chaine:
4          if c in ...:
5              dico[c] = dico[c] + 1
6          else:
7              ...
8      liste_res = ...
9      for cle in dico:
10         liste_res....
11     return ...
```

5. Recopier et compléter le code de la fonction `liste_occurrences` ci-dessus.

Pour élaborer l'arbre du codage de Huffman, on doit regrouper les nœuds en prenant toujours les deux noeuds de plus faible nombre d'occurrences. On va donc dans un premier temps faire un tri par insertion de la liste des tuples `(c, nb_occu)`, par ordre croissant de nombre d'occurrences.

On rappelle que la méthode `insert` de la classe list, utilisable avec la syntaxe : `l.insert(i, ele)`, permet d'insérer à la position `i` l'élément `ele` dans la liste `l`.

Par exemple :
```
>>> l = [0, 2, 4]
>>> l.insert(1, 8)
>>> l
[0, 8, 2, 4]
```

Le début de la fonction `tri_liste` est donné ci-dessous :

```
1  def tri_liste(liste_a_trier):
2      liste_triee = []
3      for i in range(0, ...):
4          element = liste_a_trier[i]
5          ...
6          while (j < len(liste_triee) and
7                 element[1] >= liste_triee[j][1]):
8              ...
9          liste_triee.insert(..., ...)
10     return liste_triee
```

6. Recopier et compléter le code de la fonction `tri_liste` ci-dessus.
7. Une fois la liste triée, il faut ensuite transformer la liste de tuples en liste de nœuds. Ecrire la fonction `conversion_en_noeuds` qui convertit une liste de tuples en liste de nœuds.

Afin de simplifier l'élaboration de l'arbre du codage de Huffman, on doit créer une fonction qui insère un nouveau nœud dans une liste de nœuds déjà triée par ordre croissant sur le nombre d'occurrences (attribut `nb_occu`).

```
1  def insere_noeud(noeud, liste_noeud):
2      j = 0
3      while j < len(liste_noeud) and ... > ...:
4          ...
5      liste_noeud.insert(..., ...)
```

8. Recopier et compléter le code de la fonction `insere_noeud` qui modifie la liste `liste_noeud` passée en paramètre.

On va à présent créer l'arbre du codage de Huffman à partir de la liste des nœuds triée par nombre d'occurrences croissant, avec la méthode décrite ci-dessous. Répéter les trois opérations suivantes jusqu'à ce que la liste ne soit plus composée que d'un seul nœud (la racine) :

- ⋆ extraire de la liste les deux nœuds dont le nombre d'occurrences est le plus faible ;
- ⋆ créer un nœud père permettant de regrouper ces deux nœuds ;
- ⋆ insérer ce nœud père dans la liste.

Par exemple, la Figure 1 a été obtenue en commençant par créer le nœud `'j-f'`, les nœuds `j` et `f` ayant pour nombre d'occurences `1`.

```
1 def construit_arbre(liste):
2     while ... > 1:
3         noeud1 = liste.pop(0)
4         noeud2 = liste.pop(0)
5         nom_noeud_pere = noeud1.nom + "-" + noeud2.nom
6         nb_occu_noeud_pere = ...
7         noeud_pere = Noeud(...)
8         insere_noeud(..., liste)
9     return ...
```

On rappelle que la méthode `pop` de la classe list, utilisable avec la syntaxe : `l.pop(i)`, permet de retirer et renvoyer l'élément à la position `i` dans la liste `l`. Par exemple :

```
>>> l = [0, 8, 2, 4]
>>> l.pop(1)
8
>>> l
[0, 2, 4]
```

9. Recopier et compléter le code de la fonction `construit_arbre`, qui renvoie le nœud correspondant à la racine de l'arbre.

On considère que l'on dispose d'une fonction `codage_arbre`, qui à partir d'un arbre donné en paramètre renvoie une structure telle que dans l'exemple suivant qui utilise `arb_julie`, l'arbre de la Figure 1.
Exemple :

```
>>> codage_arbre(arb_julie)
{'': '00', 'j': '0100', 'f': '0101', 'e': '011', 'l': '100',
'i': '101', 'p': '1100', 't': '11010', 'a': '11011', 'u': '111'}
```

10. Indiquer la structure de données dont il s'agit.
11. Ecrire une fonction `compresse` qui, à partir du texte et de la structure renvoyée par `codage_arbre`, renvoie la suite binaire sous forme d'une chaîne de caractères représentant le texte compressé.
    Par exemple on devrait obtenir :

```
>>> compresse('julie', {'': '00', 'j': '0100', 'f': '0101', 'e': '011', 'l': '100',
'i': '101', 'p': '1100', 't': '11010', 'a': '11011', 'u': '111'})
'0100111100101011'
```

Exercice 3 (POO, graphes et bases de données - 8 points)

Partie A

Nous avons représenté un parc d'attractions par un graphe. Les sommets de ce graphe sont des attractions. Chaque attraction a une durée (en minutes). Les arêtes de ce graphe représentent la durée (en minutes) pour aller d'une attraction à une autre. Dans ce parc d'attractions, toutes les attractions ont des noms uniques.

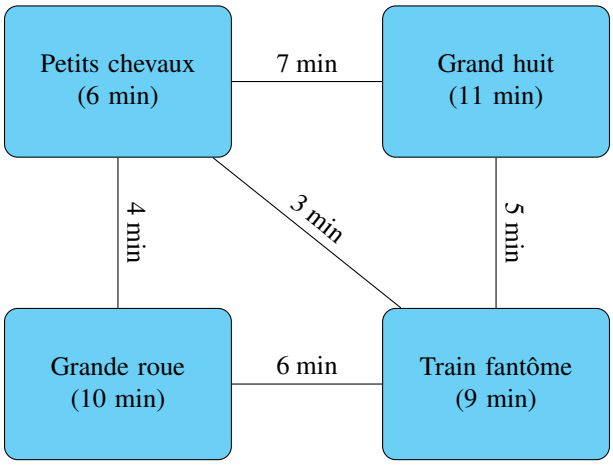


FIGURE 1 – Parc d'attractions

Les attractions sont représentées par des objets de la classe `Attraction` dont le code est donné ci-dessous.

```
class Attraction:
    def __init__(self, nom, duree):
        self.nom = nom
        self.duree = duree
        self.voisines = []
```

Le graphe précédent peut être représenté, d'une façon incomplète, en langage Python ainsi :

```
1 a1 = Attraction("Grand huit", 11)
2 a2 = Attraction("Petits chevaux", 6)
3 a3 = Attraction("Train fantôme", 9)
4 a4 = Attraction("Grande roue", 10)
5 a1.voisines = [(a2,7), (a3,5)]
6 a2.voisines = [(a1,7), (a3,3), (a4,4)]
7 a3.voisines = [(a1,5), (a2,3), (a4,6)]
8 a4.voisines = ...
```

Par mesure de sécurité, les gérants du parc d'attractions ont ralenti la vitesse de rotation de la grande roue. Sa durée est maintenant de 12 minutes.

1. En considérant la modélisation du parc d'attractions ci-dessus, écrire une ligne de code permettant de faire cette modification.
2. Donner et expliquer la valeur de l'expression `a2.voisines[2][1]`.
3. Expliquer la ligne 7 de ce code.
4. Recopier et compléter la ligne 8 de ce code.
5. Expliquer pourquoi cette modélisation du parc d'attractions est réalisée avec un graphe non orienté.

Pour faciliter la gestion du parc d'attractions, ses dirigeants proposent aux usagers des balades dans le parc. Une *balade* est un chemin du graphe représentant le parc d'attractions. Les usagers choisissant une balade doivent faire les attractions dans l'ordre de parcours du chemin. La durée d'une balade est la durée totale pour parcourir la balade, c'est-à-dire la somme des durées de ses sommets et de ses arêtes.

En langage Python, on modélise une balade par un tableau de sommets du graphe. Par exemple, le tableau `[a1, a2, a3, a1, a3]` est une balade du graphe précédent.

6. Calculer la durée en minutes de la balade représentée par le tableau `[a1, a2, a3]` et expliquer le calcul effectué en une phrase.

7. Expliquer pourquoi le tableau `[a2, a1, a4, a3]` n'est pas une balade du parc d'attractions.

On considère qu'il est possible de comparer des objets de la classe `Attraction` entre eux à l'aide de l'opérateur ==.

8. Ecrire une fonction `sont_voisines` qui prend comme arguments deux attractions de la classe `Attraction` et qui renvoie `True` si ces deux attractions sont voisines et `False` sinon.
9. Ecrire une fonction `est_balade` qui prend comme argument un tableau de sommets de type `Attraction` et qui renvoie `True` si ce tableau est une balade et `False` sinon.

Les gérants du parc d'attractions souhaitent automatiser la création de balades, de telle sorte que désormais chaque attraction apparaisse au maximum une fois dans la balade. Pour cela, ils proposent de faire un parcours de graphe à partir d'une des attractions du parc, avec un tableau pouvant représenter une balade en paramètre. Pendant le parcours du graphe, si une attraction est atteignable depuis la dernière attraction placée dans la balade, alors elle est ajoutée à la balade.

Le code suivant est proposé :

```
def parcours(attr, deja_vues, balade, nb):
    if not attr.nom in deja_vues:
        deja_vues[attr.nom] = True
        if nb == 0 or sont_voisines(attr, balade[nb-1]):
            balade[nb] = attr
            nb = nb + 1
        for voisine in attr.voisines:
            nb = parcours(voisine[0], deja_vues, balade, nb)
    return nb
```

10. Donner le type de parcours effectué par la fonction `parcours` ci-dessus.

Chaque attraction apparaît au maximum une fois dans une balade. Ainsi, un tableau représentant la balade peut être initialisé à `[None, None, None, None]` si le parc d'attractions n'a que quatre attractions. Si à l'issue du parcours, les attractions n'ont pas été toutes utilisées, il sera possible de créer une copie partielle du tableau contenant uniquement les éléments différents de `None`.

11. Déterminer ce que contient le tableau `balade` après l'exécution du code ci-dessous, en utilisant les variables `a1`, `a2`, `a3` et `a4` :

```
>>> balade = [ None for _ in range(4) ]
>>> parcours(a4, {}, balade, 0)
```

12. Déterminer maintenant ce que contient le tableau `tableau` après l'exécution du code ci-dessous :

```
>>> a2.voisines = [(a1,7), (a3,3)]
>>> a4.voisines = [(a3,6)]
>>> tableau = [ None for _ in range(4) ]
>>> parcours(a3, {}, tableau, 0)
```

13. Déduire des appels à la fonction `parcours` le nom de la structure de données utilisée pour la variable `deja_vues` et expliquer en une phrase son rôle.

Partie B

Dans cette partie de l'exercice, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT`, `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT`, `ORDER BY`.

Les visiteurs qui sont d'accord reçoivent un bracelet magnétique à l'entrée du parc d'attractions. Ce bracelet permet de les identifier et de les prendre en photos à différents points clés des attractions. Ces photos leur sont ensuite proposées à la vente. Le système est calibré pour ne pas prendre de photos des personnes ne le souhaitant pas. Les données personnelles associées sont stockées en France et les utilisateurs disposent, conformément à la loi, d'un droit de consultation, de retrait et de rectification.

Pour gérer ces photos et leur vente, le parc d'attractions utilise une base de données. La Figure 2 présente une représentation des trois relations de cette base dont les clés primaires sont les attributs soulignés, appelés id dans chaque relation et dont les clés étrangères sont précédées d'un caractère `#`. Pour chaque attribut est indiqué le nom de l'attribut, et son type après le symbole : le type int représente des entiers, le type text des chaînes de caractères et le type float des nombres flottants.

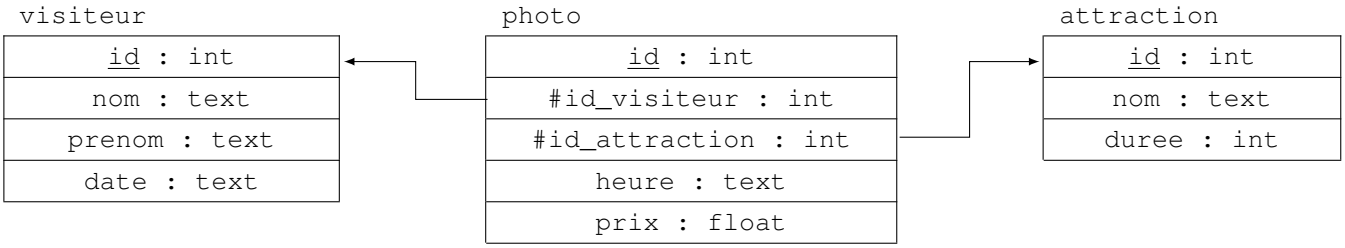


FIGURE 2 – Représentation des relations de la base de données utilisée

L'attribut `date` de la relation `visiteur` est une chaîne de caractères au format `'année-mois-jour'`, l'année étant écrite sur 4 chiffres, le mois sur 2 chiffres et le jour sur 2 chiffres. Par exemple, le 1er février 2025 sera représenté par la chaîne de caractères `'2025-02-01'`. L'attribut `heure` de la relation `photo` est une chaîne de caractères au format `'heures:minutes'`, en utilisant 2 chiffres pour les heures et 2 chiffres pour les minutes. Par exemple, l'heure 5 heures 49 minutes sera représentée par `'05:49'`.

14. Expliquer ce qu'est une clé primaire, puis ce qu'est une clé étrangère.
15. Ecrire une requête en langage SQL qui permet d'obtenir les noms et prénoms des visiteurs présents le 11 janvier 2025 sans doublons.

En langage SQL, les opérateurs de comparaison classiques peuvent être utilisés pour comparer des chaînes de caractères entre elles. Par exemple, la condition `'2025-01-01' > '2024-01-01'` serait évaluée à vrai.

La fonction d'agrégation `SUM` permet de renvoyer la somme des valeurs d'un attribut. Par exemple, le code ci-dessous permet de déterminer le prix total des photos de la relation `photo` :

```
SELECT SUM(prix) FROM photo;
```

Un visiteur, Alan TURING, est venu plusieurs fois dans le parc d'attractions en 2024. À chaque visite, il a acheté toutes les photos proposées.

16. Ecrire une requête en langage SQL qui permet d'obtenir la somme totale de ce qu'Alan TURING a payé pour des photos au parc d'attractions en 2024.

Suite à un problème technique, les gérants ont utilisé la requête suivante :

```
SELECT visiteur.nom, prenom
FROM visiteur
JOIN photo ON visiteur.id = photo.id_visiteur
JOIN attraction ON attraction.id = photo.id_attraction
WHERE attraction.nom = 'Grande roue' AND heure = '12:34' AND date = '2024-07-26';
```

17. Expliquer ce qu'ils voulaient savoir.

Les gérants du parc d'attractions décident d'étoffer leur offre d'achat de photos en proposant pour un cliché plusieurs formats et supports (A5, A6, poster, porte-clé, ...).

18. Proposer des modifications de la base de données précédente pour qu'elle puisse prendre en charge cette nouvelle offre.