Math93.com

BAC NSI - Correction Amérique Nord - 2025 - Sujet 2B (Secours) - NSI

Thèmes des exercices (sur 20 points)

- Exercice 1 (6 points) : la programmation en Python et la programmation dynamique..
- Exercice 2 (6 points) : les arbres binaires et la représentation binaire. .
- Exercice 3 (8 points) : la programmation objet en langage Python, les graphes et les bases de données.
- Lien vers le sujet

Exercice 1. la programmation en Python et la programmation dynamique.. 6 points

Cet exercice porte sur : la programmation en Python et la programmation dynamique..

Partie A

1. Montrer que, pour cette justification, le nombre d'espaces nécessaires est de 8.

Corrigé

La somme des longueurs des mots `"An", "algorithm", "must", "be"` est :

len("An") + len("algorithm") + len("must") + len("be") = 2 + 9 + 4 + 2 = 17 caractères.

On souhaite atteindre une ligne de 25 caractères.
Il faut donc : 25 − 17 = 8 espaces supplémentaires.

2. Déterminer la seule proposition qui respecte les règles d'alignement justifié pour une ligne de 25 caractères.

Corrigé

Il y a 3 emplacements inter-mots (entre les 4 mots), donc :
$8 = 3 \times 2 + 2$ Donc 2 espaces par emplacement avec reste 2 à répartir une par une de gauche à droite.
On mettra 3 espaces pour les deux premiers emplacements, et 2 pour le dernier.
Cela correspond à la proposition : `An--algorithm--must-be`

3. Compléter la ligne 5 de la fonction `ajout_espace`.

Corrigé

L'instruction correcte est :

```
assert nb_caracteres + len(liste_mots) - 1 <= justification
```

Cela garantit qu'il reste assez de place pour les espaces minimaux (un entre chaque mot).

4. Compléter les lignes 8, 14 et 17 de la fonction `ajout_espace`.

```
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 + len(liste_mots) - 1 <= justification
6      nb_espace_total = justification - nb_caracteres
7      if nb_mots == 1:
8          return liste_mots[0] + " " * 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 + " " * (q + 1) + liste_mots[i]
15         for i in range(r + 1, nb_mots):
16             reponse = reponse + " " * q + liste_mots[i]
17         return reponse
```

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.

Corrigé

Une façon de faire consisterait à placer sur la première ligne le plus de mots possibles, puis deffectuer la même chose pour les lignes suivantes tant quil reste des mots à placer.

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.

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

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.

- 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 est de proposer un découpage minimisant ce coût.

7. Compléter le tableau pour le découpage donné :

Corrigé

On utilise une justification de 15 caractères.

- Ligne 2 (mots 2 à 4 : `must be`) : longueur des mots = 4 + 2 = 6, un espace = 1, total = 7 il faut 8 espaces supplémentaires. Coût : $8^2 = 64$
- Ligne 3 (mots 4 à 7 : `seen to be`) : longueur des mots = 4 + 2 + 2 = 8, deux espaces = 2, total = 10 il faut 5 espaces supplémentaires. Coût : $5^2 = 25$
- Ligne 4 (mot 7 à 8 : `believed`) : longueur = 8, aucun espace il faut 7 espaces supplémentaires. Coût : $7^2 = 49$

Coût total = 9 + 64 + 25 + 49 = $\boxed{147}$

8. Écrire le code de la fonction `cout`.

```
def cout(i: int, j: int, liste_mots: list[str], justification: int) -> int:
    nb_mots = j - i
    nb_caracteres = sum(len(mot) for mot in liste_mots[i:j])
    nb_espaces = nb_mots - 1
    longueur_totale = nb_caracteres + nb_espaces
    if longueur_totale > justification:
        return 1000000
    espaces_supplementaires = justification - longueur_totale
    return espaces_supplementaires ** 2
```

9. Peut-on tester toutes les possibilités pour $n \geq 50$?

Corrigé

Non, ce n'est pas raisonnable. Il y a 2^{n-1} façons possibles de revenir à la ligne ou non entre chaque mot. Pour $n = 50$, cela représente plus de 10^{14} cas à évaluer, ce qui est bien trop long à calculer exhaustivement.

10. Donner l'ordre de grandeur du nombre d'appels à la fonction `cout` lors de l'exécution de `justifie_dynamique`

Corrigé

L'appel à `cout` se fait à chaque itération de deux boucles imbriquées : pour chaque i, on teste tous les $j > i$ jusqu'à n. Cela donne un nombre total d'appels proportionnel à la

somme des $n-i$, soit un ordre de grandeur en $\boxed{\mathcal{O}(n^2)}$.

11. Relation entre les éléments de la liste `cout_mini`.

Corrigé

Pour j compris entre $i+1$ et n exclu on a :

```
cout_mini[j] = mininmun(cout_mini[j] + cout(i, j, liste_mots,
justification))
```

12. Proposer une version de `justifie_dynamique` qui retourne aussi le coût inesthétique.

```
def justifie_dynamique(liste_mots: list[str], justification: int)
-> tuple[list[tuple[int, int]], int]:
    n = len(liste_mots)
    cout_mini = [0] * n
    indice_retour = [0] * n
    for i in range(n - 1, -1, -1):
        cout_mini[i] = cout(i, n, liste_mots, justification)
        indice_retour[i] = n
        for j in range(i + 1, n):
            total = cout(i, j, liste_mots, justification) + cout_mini[j]
            if total < cout_mini[i]:
                cout_mini[i] = total
                indice_retour[i] = j
    decoupage = []
    k = 0
    while k < n:
        decoupage.append((k, indice_retour[k]))
        k = indice_retour[k]
    return decoupage, cout_mini[0]
```

Exercice 2. les arbres binaires et la représentation binaire. . 6 points

Cet exercice porte sur : les arbres binaires et la représentation binaire. .

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

Corrigé

Une feuille est un nœud sans fils. Exemple : le caractère `p` est une feuille. La racine est le nœud en haut de l'arbre, ici nommé `j-f-e-l-a-p-i-t-u- -1`.

2. Donner la profondeur du nœud correspondant au caractère `p` et son code binaire associé.

Corrigé

Le caractère `p` est à 4 arêtes de la racine, donc sa profondeur est **4**. Son code binaire est **1100** (droite, droite, gauche, gauche).

3. Expliquer l'intérêt de placer les caractères les plus fréquents à faible profondeur dans l'arbre.

Corrigé

Les caractères fréquents ayant un code plus court, cela réduit la longueur totale du message compressé. C'est l'objectif du codage de Huffman : **minimiser la taille du message binaire** en attribuant les codes les plus courts aux caractères les plus fréquents.

4. Compléter les lignes manquantes de la classe `Noeud`.

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

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

5. Compléter la fonction `liste_occurrences`.

```
def liste_occurrences(chaine):
    dico = {}
    for c in chaine:
        if c in dico:
            dico[c] += 1
        else:
            dico[c] = 1
    res = []
    for c in dico:
        res.append((c, dico[c]))
    return res
```

6. Compléter la fonction `tri_liste`.

```
def tri_liste(liste_a_trier):
    liste_triee = []
    for i in range(len(liste_a_trier)):
        element = liste_a_trier[i]
        j = 0
        while j < len(liste_triee) and element[1] >= liste_triee[j][1]:
            j += 1
        liste_triee.insert(j, element)
    return liste_triee
```

7. Écrire la fonction `conversion_en_noeuds`.

```
def conversion_en_noeuds(liste_tuples):
    res = []
    for (lettre, occu) in liste_tuples:
        res.append(Noeud(lettre, occu, None, None))
    return res
```

8. Compléter la fonction `insere_noeud`.

```
def insere_noeud(noeud, liste_noeud):
    j = 0
    while j < len(liste_noeud)
    and noeud.nb_occu >= liste_noeud[j].nb_occu:
        j += 1
    liste_noeud.insert(j, noeud)
```

9. Compléter la fonction `construit_arbre`.

```
def construit_arbre(liste):
    while len(liste) > 1:
        noeud1 = liste.pop(0)
        noeud2 = liste.pop(0)
        nom_noeud_pere = noeud1.nom + '-' + noeud2.nom
        nb_occu_noeud_pere = noeud1.nb_occu + noeud2.nb_occu
        noeud_pere =
          Noeud(nom_noeud_pere, nb_occu_noeud_pere, noeud1, noeud2)
        insere_noeud(noeud_pere, liste)
    return liste[0]
```

10. Indiquer la structure de données utilisée par `codage_arbre`.

Corrigé

Il s'agit d'un **dictionnaire Python** (`dict`) dont les clés sont les caractères et les valeurs sont leurs codes binaires associés.

11. Écrire la fonction `compresse`.

```
def compresse(texte: str, codage: dict) -> str:
    resultat = ""
    for c in texte:
        resultat += codage[c]
    return resultat
```

Exercice 3. la programmation objet en langage Python, les graphes et les bases de données. 8 points

Cet exercice porte sur : la programmation objet en langage Python, les graphes et les bases de données.

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.

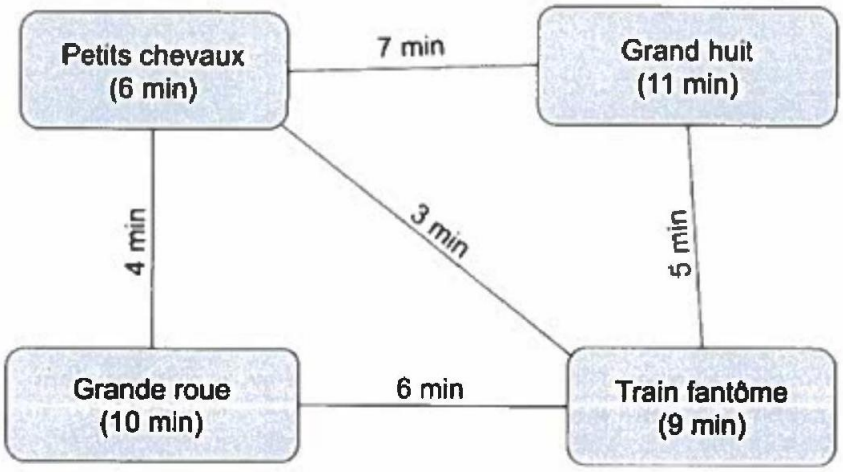

Figure 1. Parc d'attractions

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 = ...
```

1. Ligne de code pour modifier la durée de la Grande roue :

Corrigé

```
a4.duree = 12
```

2. Donner et expliquer la valeur de `a2.voisines[2][1]`.

Corrigé

`a2.voisines[2]` donne le 3e couple de la liste des voisines de a2, soit `(a4, 4)`.
`a2.voisines[2][1]` donne donc $\boxed{4}$, c'est la durée du trajet de `a2` à `a4`.

3. Expliquer la ligne 7 du code.

Corrigé

La ligne : `a3.voisines = [(a1,5), (a2,3), (a4,6)]` signifie que `a3` (Train Fantome) est voisine de :

- `a1` (Grand Huit) avec un trajet de 5 minutes,
- de `a2` (Petits chevaux) avec 3 minutes
- et de `a4` (grande roue) avec 6 minutes.

4. Compléter la ligne 8 du code.

Corrigé

```
a4.voisines = [(a2, 4), (a3, 6)]
```

5. Justifier l'usage d'un graphe non orienté.

Corrigé

Le graphe est non orienté car les trajets entre deux attractions sont symétriques : aller de `a1` à `a2` prend le même temps que de `a2` à `a1`, donc les arêtes n'ont pas de direction.

6. Calcul de la durée de la balade `[a1, a2, a3]`.

Corrigé

- Durée de `a1` = 11,
- trajet `a1 --> a2` = 7,
- durée de `a2` = 6,
- trajet `a2 --> a3` = 3,
- durée de `a3` = 9

Total : $11+7+6+3+9=\boxed{36}$ minutes.

7. Pourquoi `[a2, a1, a4, a3]` n'est pas une balade.

Corrigé

Il n'existe pas d'arête directe entre `a1` et `a4`, donc ce trajet n'existe pas dans le graphe. Ce n'est pas une balade valide.

8. Écrire la fonction `sont_voisines`.

```
def sont_voisines(a, b):
    for (voisine, _) in a.voisines:
        if voisine == b:
            return True
    return False

# ou
def sont_voisines(attraction1, attraction2):
for couple in attraction1.voisines:
    if couple[0] == attraction2:
        return True
    return False

```

9. Écrire la fonction `est_balade`.

```
def est_balade(liste_attractions):
    for i in range(len(liste_attractions) - 1):
        if not sont_voisines(liste_attractions[i], liste_attractions[i+1]):
            return False
    return True
```

10. Type de parcours effectué par la fonction `parcours`.

Corrigé

Il s'agit d'un **parcours en profondeur** (depth-first search), car on explore récursivement chaque voisine avant de revenir au niveau précédent.

11. Résultat du code `parcours(a4, {}, balade, 0)`.

Corrigé

Pour un parcours en profondeur on obtient donc : `[a4,a2,a1,a3]`

Pour un parcours en largeur on aurait : `[a4,a2,a3,a1]`

12. Résultat du second code avec `a2.voisines = [(a1,7), (a3,3)]`,`a4.voisines = [(a3,6)]`

Corrigé

Donc `tableau` contient :

`[a3,a1,a2,None]`

13. Structure de `deja_vues` et rôle.

Corrigé

`deja_vues` est un **dictionnaire** associant un nom d'attraction (chaîne) à un booléen. Il permet d'éviter de visiter plusieurs fois la même attraction.

Partie B

14. Clé primaire et clé étrangère.

Corrigé

Une **clé primaire** est un attribut unique permettant d'identifier chaque ligne d'une table.
Une **clé étrangère** est un attribut qui fait référence à la clé primaire d'une autre table pour établir une relation entre les deux tables.

15. Requête SQL pour visiteurs présents le 11 janvier 2025 sans doublons.

```sql
SELECT DISTINCT nom, prenom
FROM visiteur
WHERE date = '2025-01-11';
```

16. Requête SQL pour les achats de Alan TURING en 2024.

```sql
SELECT SUM(prix)
FROM visiteur JOIN photo
  ON visiteur.id = photo.id_visiteur
WHERE nom = 'TURING'
  AND prenom = 'Alan'
  AND date >= '2024-01-01'
  AND date <= '2024-12-31';
```

17. Objectif de la requête des gérants.

Corrigé

Ils veulent connaître l'identité (nom, prénom) des personnes présentes sur la photo prise à l'attraction " Grande roue " le **26 juillet 2024 à 12 :34**.

18. Proposition de modification de la base.

Corrigé

Ajouter une nouvelle table `format` avec les colonnes :

- `id` (clé primaire),
- `nom` (A5, poster...),
- `prix`,
- `support` (papier, porte-clé...).

Ajouter une relation `photo_format` avec `#id_photo`, `#id_format`.