# Exercice 1 (6 points)

Correction

| Question | Niveau | Contenu | Solution |
| --- | --- | --- | --- |
| 1 | 1 | Passer de la représentation d'une base dans une autre. | 01100001 |
| 2 | 1 | Construire un tableau par compréhension. | [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1] |
| 3 | 2 | Itérer sur les éléments d'un tableau. Construire une entrée de dictionnaire | 14 if x in nb\_occ: 15 nb\_occ[x] = nb\_occ[x] + 1 16 else: 17 nb\_occ[x] = 1 |
| 4 | 2 | Itérer sur les éléments d'un dictionnaire. | 14 for cle in dict.keys(): 15 if dict[cle] > valeur\_max: 16 valeur\_max = dict[cle] 17 cle\_max = cle 18 return cle\_max |
| 5 | 1 | Mise au point des programmes. Gestion des bugs. | L’erreur se situe à l’intersection de la première ligne et de la seconde colonne. |
| 6 | 3 | Utiliser des tableaux de tableaux pour représenter des matrices : notation a[i][j]. | 9 def erreur\_colonne(mat): 10 for num\_col in range(3): 11 if sum([mat[num\_lig][num\_col] for num\_lig in range(3)]) % 2 != 0: 12 return num\_col  Autre version possible |
| 7 | 1 |  | Le code du tableau le plus proche est 1110000, qui ne diffère que par son second bit du code reçu.  Il correspond au mot 1000. |
| 8 | 2 |  | 16 # Copie du code recu créée par compréhension 17 code = [bit for bit in code\_recu]  18 for indice in range(7): 19 # Inversion du bit d'indice courant 20 code[indice] = (code[indice] + 1) % 2  21 if code in liste\_code\_correct: 22 return code 23 else: 24 # Réinit. de bit d'indice courant 25 code[indice] = ((code[indice] + 1) % 2) |
| 9 | 1 | Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits. |  |
| 10 | 3 | Algorithmes sur les arbres binaires. Parcourir un arbre. Récursivité. Vocabulaire de la programmation objet : classes, attributs, méthodes, objets. | 11 if code[i] == 0: 12 return decode(arbre.gauche, code, i+1) 13 if code[i] == 1: 14 return decode(arbre.droit, code, i+1) |

# Exercice 2 (6 points)

Correction

| Question | Niveau | Contenu | Solution |
| --- | --- | --- | --- |
| 1 | 1 | comprendre une situation/un problème | On mange successivement les bonbons d’indices 0, 3, 6, 2, 7, 5, 1 et le dernier bonbon possède l’indice 4.  Figure 1. (correction)  Figure 1. (correction) |
| 2 | 1 | interpréter une erreur de programmation | Une erreur de type NameError indique une variable de nom inconnu. Bob a saisi true que Python interprète comme un nom de variable. Il faut taper True afin d’utiliser le booléen. |
| 3 | 3 | compléter une fonction Python simple | 1 def dernier(n):  2 collier = [True for i in range(n)] # création de la liste collier  3 indice = 0  4 collier[indice] = False  5 for etape in range(n-1):  6 nb\_bonbons\_vus = 0  7 while nb\_bonbons\_vus != 3:  8 indice += 1  9 if indice >= n: 10 indice = 0 11 if collier[indice]: 12 nb\_bonbons\_vus += 1 13 collier[indice] = False 14 return indice |
| 4 | 1 | connaître les structures de données | Une file est une structure de données linéaire suivant le modèle « *Premier entré, Premier sorti* ». |
| 5 | 1 | faire fonctionner un algorithme à la main | Initialement la file est vide. On enfile ensuite les valeurs de [0, 1, 2, 3, 4]. 0 est en tête de la file et 4 à la queue.  On défile donc successivement 0, puis 1 et enfin 2.  Seules les deux dernières valeurs sont enfilées à nouveau.  On a donc finalement : (Tête) 3 4 1 2 (Queue). |
| 6 | 2 | compléter une fonction Python simple | def dernier\_file(n):  collier = File()  for i in range(n):   collier.enfile(i)   for tour in range(n - 1):   collier.defile()  collier.enfile(collier.defile())   collier.enfile(collier.defile())   return collier.defile() |
| 7 | 1 | connaître le vocabulaire de la POO | Ces variables sont des *attributs*. |
| 8 | 2 | faire fonctionner un algorithme à la main | Le succ de zero est un. La variable a contient sa valeur, donc a = 1.  Le pred du succ du succ de un est l’objet deux. La variable b contient sa valeur, donc b = 2. |
| 9 | 2 | compléter une fonction Python | def creer\_collier(n):  premier = Bonbon(0)  actuel = premier  for i in range(1, n):  nouveau = Bonbon(i)   actuel.succ = nouveau   nouveau.pred = actuel   actuel = nouveau  actuel.succ = premier   premier.pred = actuel   return premier |
| 10 | 2 | faire fonctionner un algorithme à la main | On travaille sur un collier de 4 bonbons, numérotés de 0 à 3.  Les opérations du code ont pour effet de supprimer le premier bonbon (de valeur 0). La variable bonbon est le succ de premier : il s’agit du bonbon de valeur 1.  On a donc :  Figure 2. (correction)  Figure 2. (correction) |
| 11 | 1 | comprendre une situation/problème | Il s’agit de la proposition C : bonbon.valeur == bonbon.succ.valeur. |
| 12 | 2 | compléter une fonction Python complexe | def dernier\_chaine(n):  bonbon = creer\_collier(n)  while bonbon.valeur != bonbon.succ.valeur:   bonbon.pred.succ = bonbon.succ   bonbon.succ.pred = bonbon.pred   bonbon = bonbon.succ.succ.succ   return bonbon.valeur |

# Exercice 3 (8 points)

Correction

| Question | Niveau | Contenu | Solution |
| --- | --- | --- | --- |
| 1 | 1 | Modèle relationnel | Il ne permet pas d’identifier de façon unique chaque enregistrement de la relation (sinon, cela ne permettrait qu’une seule réservation par vol, donc un seul passager par avion). |
| 2 | 1 | Modèle relationnel | * le couple (id\_vol, id\_passager); * ou une proposition de nouvelle colonne id\_reservation. |
| 3 | 1 | Modèle relationnel | Elle indique qu’une colonne dans une table fait référence à la clé primaire d’une autre table. Cela permet de créer des liens entre les données de différentes tables. |
| 4 | 1 | Langage SQL | AI0015, AI0258, AI0292 |
| 5 | 2 | Langage SQL | SELECT ville FROM aeroport JOIN vol ON vol.aeroport\_arr = id\_aeroport WHERE aeroport\_dep = 'CDG'; |
| 6 | 1 | Langage SQL | UPDATE passager SET d\_totale = 16 WHERE id\_passager = 5; |
| 7 | 2 | Base de données relationnelle et langage SQL | L’identifiant du vol n’est pas correct.  Remarque : il existe déjà ce qui va avoir pour effet le rejet de la requête puisqu’il s’agit d’une clé primaire.  INSERT INTO vol (id\_vol, aeroport\_dep, aeroport\_arr, distance) VALUES ('AI1000', 'CDG', 'YUL', 6); |
| 8 | 1 | Dictionnaire | 10 |
| 9 | 2 | Programmation et dictionnaire | 1 def vol\_direct(graphe, ville1, ville2): 2 return ville2 in graphe[ville1] 3 # ou : return ville1 in graphe[ville2] |
| 10 | 3 | Programmation et dictionnaire | 1 def liste\_villes\_proches(graphe, ville, d\_max): 2 resultat = list() 3 for voisine in graphe[ville]: 4 if graphe[ville][voisine] <= d\_max: 5 resultat.append(voisine) 6 return resultat |
| 11 | 1 | Graphes | Figure 2. Graphe non orienté pondéré correspondant au réseau aérien de la compagnie Droidevant.  Figure 2. Graphe non orienté pondéré correspondant au réseau aérien de la compagnie *Droidevant*. |
| 12 | 1 | Graphes | Proposition A |
| 13 | 1 | Récursivité | À la ligne 12 la fonction parcours s’appelle elle-même, la fonction parcours peut donc être qualifiée de fonction récursive. |
| 14 | 3 | Algorithmique - Récursivité | La variable visitees1 contient ['W', 'P', 'T', 'B', 'S']. La variable visitees2 contient ['W', 'P', 'B']. |
| 15 | 2 | Parcours d'un graphe | Proposition C |
| 16 | 2 | Programmation | 1 def est\_connexe(graphe): 2 """Vérifie si un graphe est connexe.""" 3 depart = ville\_arbitraire(graphe) 4 visitees = []  5 parcours(graphe, visitees, depart) 6 return len(visitees) == len(graphe) |
| 17 | 3 | Algorithmique - Récursivité - Parcours d'un graphe | ['W', 'P', 'T', 'B'] 25 ['W', 'P', 'S', 'T', 'B'] 40 |
| 18 | 3 | Algorithmique - Récursivité - Parcours d'un graphe | Cet appel affiche tous les chemins du graphe allant de ville à arrivee sans repasser par un même sommet ainsi que le poids de ces chemins, c’est-à-dire la somme des poids des arêtes de ces chemins. |