BACCALAURÉAT BLANC n°2   
CORRECTION

**EXERCICE 1 (7 points)**

**Partie A : Recherche par force brute**

1. La recherche par « force brute », aussi appelée recherche *exhaustive* consiste à résoudre un problème en énumérant *toutes* les solutions possibles (afin de déterminer la meilleure).
2. Il y a 6 chemins possibles (on donne aussi les déplacements correspondants : D pour Droite et B pour Gauche)

* (0, 0), (0, 1), (0, 2), (1, 2), (2, 2) 🡪 déplacement DDBB
* (0, 0), (0, 1), (1, 1), (1, 2), (2, 2) 🡪 déplacement DBDB
* (0, 0), (0, 1), (1, 1), (2, 1), (2, 2) 🡪 déplacement DBBD
* (0, 0), (1, 0), (1, 1), (1, 2), (2, 2) 🡪 déplacement BDDB
* (0, 0), (1, 0), (1, 1), (2, 1), (2, 2) 🡪 déplacement BDBD
* (0, 0), (1, 0), (2, 0), (2, 1), (2, 2) 🡪 déplacement BBDD

Chaque chemin consiste à se déplacer 2 fois vers la droite et 2 fois vers le bas, ce qui fait parcourir 4 cases pour arriver dans la case en bas à droite. En ajoutant la case initiale (0,0), chaque chemin a une longueur égale à 5.

1. La fonction s’appelle elle-même (lignes 11 et 14) donc elle est récursive.

|  |
| --- |
| **def** liste\_chemins(n):  resultats = []  parcours(**0**, **0**, [(0, 0)], resultats, n)  **return** resultats  **def** parcours(i, j, chemin, resultats, n):  **if** i == **n - 1 and** j == **n - 1**: # Si on atteint la case en bas à droite  resultats.append(chemin) # Ajouter le chemin actuel aux résultats  **else**:  **if** i < n - 1: # Se déplacer vers le bas si possible  parcours(i + 1, j, chemin + [(**i + 1**, **j**)], resultats, n)  **if** j < n - 1: # Se déplacer vers la droite si possible  parcours(**i**, **j + 1**, **chemin + [(i, j + 1)]**, resultats, n) |

|  |
| --- |
| **def** somme\_chemin(chemin, tab):  s = 0  **for** c **in** chemin:  s = s + tab[c[0]][c[1]]  **return** s |

|  |
| --- |
| **def** somme\_max\_force\_brute(tab):  chemins = liste\_chemins(len(tab))  somme\_max = 0  **for** chemin **in** chemins:  s = somme\_chemin(chemin, tab)  **if** s > somme\_max:  somme\_max = s  **return** somme\_max |

1. Le nombre de chemins à considérer devient vite extrêmement grand dès que la taille du tableau augmente. La stratégie de force brute n’est donc pas applicable dès que n est trop grand.

**Partie B : Recherche par méthode gloutonne**

1. Un algorithme glouton consiste à résoudre un problème étape par étape, et à chaque étape le meilleur choix local est effectué. Ce choix n’est jamais remis en cause, ce qui permet d’aboutir rapidement à une solution globale (que l’on espère bonne).

|  |
| --- |
| **def** solution\_gloutonne(tab):  n = len(tab)  somme = tab[0][0]  i = 0 # indice de ligne  j = 0 # indice de colonne  # on n’a atteint aucun des bords droit ou bas,  # on choisit la meilleure option :  **while** i < n - 1 **and** j < **n - 1**:  a\_droite = tab[i][j+1] # case de droite  en\_bas = **tab[i+1][j]** # case du dessous  **if** a\_droite >= en\_bas:  somme = **somme + a\_droite**  j = **j + 1**  **else**:  **somme = somme + en\_bas**  **i = i + 1**  # on a atteint le bas, on va à droite :  **if** i == n - 1:  **for** k **in** range(**j+1**, **n**):  somme = somme + tab[i][k] # ou tab[n-1][k]  # ou alors on a atteint le bord droit, on descend :  **else**:  **for** k **in** **range(i+1, n)**:  **somme = somme + tab[k][j]** # ou tab[k][n-1]  **return** somme |

1. En remplaçant par exemple la valeur en haut à droite du tableau T donné en exemple par 50, la solution gloutonne reste identique (chemin en gras de somme 20) mais on peut trouver un meilleur chemin : celui allant trois fois à droite puis trois fois vers le bas (chemin grisé), dont la somme vaut 61.

|  |  |  |  |
| --- | --- | --- | --- |
| **4** | 1 | 3 | 50 |
| **2** | 0 | 2 | 1 |
| **7** | 1 | 2 | 1 |
| **2** | **1** | **3** | **1** |

1. Si on part de la case en haut à gauche, il faut toujours effectuer déplacements vers la droite et déplacements vers le bas pour arriver en base à droite. Cela correspond à cases, et en ajoutant la case (0, 0), on obtient toujours un chemin de longueur .
2. À chaque étape, l’algorithme glouton consiste à faire un choix pour ajouter une case au chemin construit. Comme un chemin est de longueur , il y a donc étapes à effectuer (ou si on ne compte pas la case initiale). Chaque étape étant composée d’opérations en temps constant, le coût de la fonction solution\_gloutonne est de l’ordre de , donc de l’ordre de  : son coût est linéaire.

**Partie C : Recherche par programmation dynamique**

1. La programmation dynamique consiste à chercher la solution à un problème en décomposant ce problème en sous-problèmes, à résoudre les sous-problèmes du plus petit au plus grand en stockant les résultats intermédiaires.
2. Voici le tableau complété :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 4 | 5 | 8 | **11** |
| s = | 6 | 6 | 10 | 12 |
| 13 | **14** | 16 | 17 |
|  | 15 | 16 | **19** | 20 |

1. La case (0,0) de s est égale à tab[0][0]. Les cases de la première colonne de s correspondent aux sommes maximales des chemins de (0, 0) à (i, 0), donc à se déplacer toujours vers le bas. Il suffit de cumuler les valeurs de la première colonne de tab pour obtenir la première colonne de s. De la même manière, il suffit de cumuler les valeurs de la première ligne de tab pour obtenir celles de s.  
   Si on veut traduire ces constatations par des formules, cela donne :

* pour i différent de 0 : s[i][0] = s[i-1][0] + tab[i][0]
* pour j different de 0 : s[0][j] = s[0][j-1] + tab[0][j]

1. Pour arriver en case (i, j), il y a deux possibilités : soit on vient du dessus, c’est-à-dire de la case (i-1, j), soit on vient de la gauche, c’est-à-dire de la (i, j-1). Comme on cherche la somme maximale pour arriver en (i,j) (c’est-à-dire s[i][j]), il faut choisir parmi les deux cases (du dessus et de gauche) celle qui contient la plus grande somme (c’est-à-dire max(s[i-1][j], s[i][j-1])). Il suffit alors d’ajouter la valeur de la dernière case (c’est-à-dire tab[i][j]) à cette somme et on obtient la somme maximale pour arriver en (i, j). C’est exactement la formule indiquée :

s[i][j] = tab[i][j] + max(s[i-1][j], s[i][j-1])

|  |
| --- |
| **def** somme\_max\_dynamique(tab):  # création et initialisation du tableau  n = len(tab)  s = [[None] \* n **for** i **in** range(n)]  s[0][0] = tab[**0**][**0**]  # remplissage du reste de la première ligne  **for** j **in** range(1, n):  s[0][j] = **s[0][j-1] + tab[0][j]**  # remplissage du reste de la première colonne  **for** i **in** **range(1, n)**:  s[i][0] = **s[i-1][0] + tab[i][0]**  # remplissage du reste du tableau  **for** i **in** **range(1, n)**:  **for** j **in** **range(1, n)**:  s[i][j] = **tab[i][j] + max(s[i-1][j], s[i][j-1])**  # renvoi de la somme maximale  **return** **s[n-1][n-1]** |

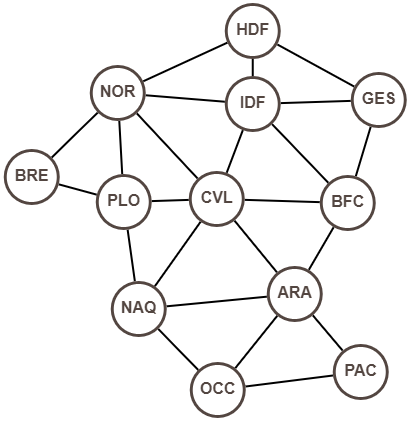
1. L’algorithme remplit successivement les cases du tableau donc il y a affectations d’une valeur à une case du tableau s dans cette fonction. Le coût de la fonction est donc bien quadratique.

Si on veut détailler les choses, il y a : affectation ligne 5 (la case en haut à gauche) ; affectations ligne 8 (les autres cases de la première ligne) ; affectations ligne 11 (les autres cases de la première colonne) ; affectations ligne 15 (les autres cases). En tout, on a donc  affectations.

**EXERCICE 2 (9 points)**

**Partie A : Création d’un parcours de visites**

1. Voici le graphe des régions :



|  |
| --- |
| **def** voisins(G, s):  **return** G[s] |

1. L’utilisation d’une file pour stocker les sommets encore à visiter a pour effet de visiter en premier le sommet qui a été ajouté en premier dans la file. Ainsi, ce sont les premiers voisins découverts qui sont visités en premier : d’abord les voisins à distance 1 du sommet de départ, puis ceux à distance 2. Il s’agit bien d’un parcours en largeur.

|  |
| --- |
| **def** sommets\_a\_distance(G, depart, n):  visites = {depart: 0} # départ est à distance 0 de lui-même  file = [depart]  sommets\_a\_proximite = [**depart**] # initialisation de la liste à renvoyer   **while** file != []:  s = file.pop(0)  **for** voisin **in** voisins(G, s):  **if** voisin **not** **in** visites:  distance\_voisin = visites[s] + **1**  **if** distance\_voisin <= **n**:  file.append(voisin)  visites[**voisin**] = **distance\_voisin**  sommets\_a\_proximite.append(voisin)  **return** sommets\_a\_proximite |

1. La recherche (d’une clé, ou d’une valeur) dans un dictionnaire se fait en temps constant alors que la recherche dans un tableau a un coût linéaire (dans le pire cas). Ainsi, tester si voisin est (ou n’est pas) dans visites est plus efficace si visites est un dictionnaire.

**Partie B : Coloration de la carte**

|  |  |
| --- | --- |
| **Sommet** | **Couleur** |
| A | Rouge |
| B | Rouge |
| C | Bleu |
| D | Vert |
| E | Vert |
| F | Rouge |

1. Les sommets sont coloriés dans l’ordre suivant :

• A, le choix est rapide : Rouge ;

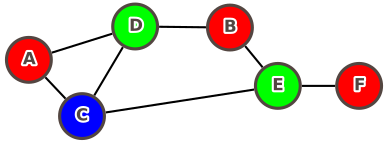
• B, le choix est également rapide : Rouge ;

• C, le Rouge est indisponible, le choix est Bleu ;

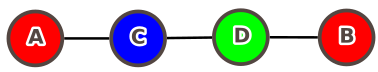
• D, Rouge et Bleu sont indisponibles, le choix suivant est Vert ;

• E, Rouge et Bleu sont indisponibles, le choix suivant est Vert ;

• F, Vert est indisponible, le premier choix Rouge est validé.



1. Voici le graphe G2 coloré :



On peut proposer une coloration avec seulement deux couleurs : A et D en rouge ; C et B en bleu.

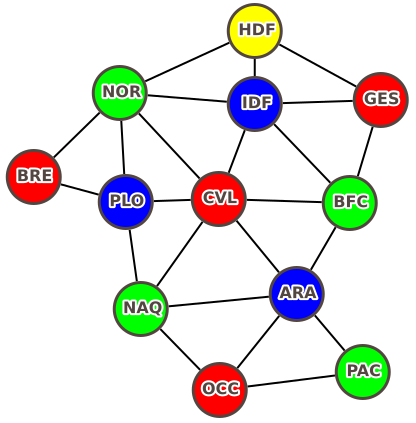
1. Voici :

|  |
| --- |
| **def** tri\_sommets(G):  sommets = [sommet **for** sommet **in** G]  **for** i **in** range(1, **len(sommets)**):  s = sommets[i]  j = i  **while** j > 0 **and** **len(G[s])** > len(G[sommets[j-1]]):  sommets[j] = sommets[**j-1**]  j = j - 1  sommets[j] = **s**  **return** sommets |

1. C’est un tri par insertion (on insère chaque élément au bon endroit dans la partie triée au début), qui a un coût quadratique (de l’ordre de , où est la taille du tableau à trier).  
   Le tri fusion est un exemple d’algorithme de tri plus efficace.
2. Voici la liste renvoyée contenant les sommets triés par degrés décroissants (le tri par insertion est stable, c’est-à-dire qu’il conserve l’ordre de départ en cas d’égalité) :  
   ['CVL', 'ARA', 'IDF', 'NOR', 'PLO', 'NAQ', 'BFC', 'OCC', 'GES', 'HDF', 'BRE', 'PAC']
3. On peut par exemple, construire un tableau de tuples de la forme (sommet, nombre de voisins) et trier ce tableau par ordre décroissant selon le second élément des tuples :

|  |
| --- |
| **def** tri\_sommets(G):  sommets = [(s, len(G[s])) **for** s **in** G]  sommets\_tries = sorted(sommets, key=**lambda** t: t[1], reverse=True)  **return** [s[0] **for** s **in** sommets\_tries] |

(Le tri implémenté par la fonction sorted est plus efficace en temps que le tri par insertion)

1. Il suffit de modifier la ligne 7 pour itérer sur itérer sur la liste triée par ordre décroissant des degrés des sommets. On remplace donc cette ligne 7 par : **for** s **in** tri\_sommets(G):
2. Les couleurs sont attribuées de la manière ci-dessous. On voit qu’il suffit de 4 couleurs.  
   *Remarque* : on aurait également utilisé 4 couleurs pour l’algorithme de coloration séquentielle du Code 1, mais la version de Welsh et Powell aurait bien donné 2 couleurs pour le graphe G2, contre 3 pour la coloration séquentielle.

|  |  |
| --- | --- |
| **Région** | **Couleur** |
| CVL | Rouge |
| ARA | Bleu |
| IDF | Bleu |
| NOR | Vert |
| PLO | Bleu |
| NAQ | Vert |
| BFC | Vert |
| OCC | Rouge |
| GES | Rouge |
| HDF | Jaune |
| BRE | Rouge |
| PAC | Vert |

**Partie C : Base de données de l’application**

1. Les avantages des SGBD sont nombreux : ils sont conçus pour optimiser l’efficacité (en temps) des requêtes, ils gèrent les autorisations d’accès, les accès concurrents et aussi la redondance des données en cas de problèmes. De plus, un SGBD relationnel s’assure en permanence du respect des contraintes d’intégrité pour que les données stockées restent cohérentes (avec le modèle relationnel).
2. Une clé primaire doit identifier de manière unique les enregistrements d’une table. Or, plusieurs lieux peuvent avoir le même nom (c’est même le cas ici), cela implique que l’attribut nom ne peut être clé primaire.
3. SELECT nom, latitude, longitude  
   FROM lieu  
   WHERE type = 'château';
4. SELECT nom   
   FROM lieu  
   WHERE type = 'musée' AND id\_region = 1  
   ORDER BY nom;
5. UPDATE lieu   
   SET nom = 'Les Étincelles du Palais de la Découverte', latitude = 48.84113, longitude = 2.27966  
   WHERE id = 6;
6. INSERT INTO lieu  
   VALUES (1253, 'La Géode', 'Paris', 48.894444, 2.388611, 'cinéma', 9);
7. On doit faire une jointure car on ne connaît pas l’identifiant de la région « Grand Est » :  
   SELECT lieu.nom  
   FROM lieu  
   JOIN region ON lieu.id\_region = region.id  
   WHERE region.nom = 'Grand Est';
8. On peut proposer le schéma relationnel suivant :  
   utilisateur(id, email, motdepasse)  
   note(#id\_utilisateur, #id\_lieu, note) où id\_utilisateur et id\_lieu font respectivement référence aux attributs id des tables utilisateur et lieu.
9. On peut construire un dictionnaire dans lequel les clés seront les identifiants des lieux et les valeurs associées seront également des dictionnaires qui stockeront à la fois la somme cumulée des notes et le nombre cumulé des notes pour ce lieu.  
   Pour construire ce dictionnaire, on parcourt la liste notes, si un lieu n’est pas encore dans le dictionnaire on l’ajoute en initialisant la somme des notes et le nombre de notes, sinon on ajoute la note à la somme et on ajoute 1 au nombre de notes déjà stockés.  
   Enfin, il suffit de renvoyer un dictionnaire dont les clés sont les lieux et les valeurs associées sont les moyennes correspondantes, calculées en divisant la somme des notes par leur nombre  
   Voici une fonction possible qui effectue cette tâche :

|  |
| --- |
| **def** moyennes(notes):  d = {}  **for** ligne **in** notes:  id\_lieu, nom, note = ligne  **if** id\_lieu **not** **in** d:  d[id\_lieu] = {'somme': note, 'nb': 1}  **else**:  d[id\_lieu]['somme'] = d[id\_lieu]['somme'] + note  d[id\_lieu]['nb'] = d[id\_lieu]['nb'] + 1  **return** {id\_lieu: d[id\_lieu]['somme'] / d[id\_lieu]['nb'] **for** id\_lieu **in** d} |

Une autre version possible qui suit le même principe mais en cumulant les notes dans une liste au fur et à mesure :

|  |
| --- |
| **def** moyennes(notes):  d = {}  **for** ligne **in** notes:  id\_lieu, nom, note = ligne  **if** id\_lieu **not** **in** d:  d[id\_lieu] = [note]  **else**:  d[id\_lieu].append(note)  print(d)  **return** {id\_lieu: sum(d[id\_lieu]) / len(d[id\_lieu]) **for** id\_lieu **in** d} |

**EXERCICE 3 (4 points)**

**Partie A**

1. C’est la proposition 2.
2. cd ../fiches
3. mkdir algorithmique
4. rm rapport.odt

**Partie B**

1. 11, 20, 32, 11, **20, 32, 11, 32, 11**
2. 11, 11, 20, 20, 32, 32, 11, 11, 32
3. Si P1 mobilise R1 et est en attente de R2 pour poursuivre, et que P2 mobilise R2 et est en attente de R1 pour poursuivre, les deux processus sont en interblocage.

**Partie C**

1. liste\_attente = [Processus(11, 4), Processus(20, 2), Processus(32, 3)]

|  |
| --- |
| **def** execute\_un\_cycle(self):  """Met à jour le reste à faire après l'exécution d'un  cycle."""  self.reste\_a\_faire = self.reste\_a\_faire - 1    **def** change\_etat(self, nouvel\_etat):  """Change l'état du processus avec la valeur passée en  paramètre."""  self.etat = nouvel\_etat    **def** est\_termine(self):  """Renvoie True si le processus est terminé, False sinon,  en se basant sur le reste à faire."""  **return** self.reste\_a\_faire == 0 |

|  |
| --- |
| **def** tourniquet(liste\_attente, quantum):  ordre\_execution = []  **while** liste\_attente != []:  # On extrait le premier processus  processus = liste\_attente.pop(0)  processus.change\_etat("En cours d'exécution")  compteur\_tourniquet = 0  **while** **not** processus.est\_termine() **and** compteur\_tourniquet < quantum:  ordre\_execution.append(processus.pid)  processus.execute\_un\_cycle()  compteur\_tourniquet = compteur\_tourniquet + 1  **if** not processus.est\_termine():  processus.change\_etat("Suspendu")  liste\_attente.append(processus)  **else**:  processus.change\_etat("Terminé")  **return** ordre\_execution |