

# 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.

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

4.

```
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)
```

5.

```
def somme_chemin(chemin, tab):
    s = 0
    for c in chemin:
        s = s + tab[c[0]][c[1]]
    return s
```

6.

```
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
```

7. 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

8. 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).
- 9.

```
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
```

10. 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

11. Si on part de la case en haut à gauche, il faut toujours effectuer  $n - 1$  déplacements vers la droite et  $n - 1$  déplacements vers le bas pour arriver en base à droite. Cela correspond à  $(n - 1) + (n - 1) = 2n - 2$  cases, et en ajoutant la case (0, 0), on obtient toujours un chemin de longueur  $2n - 1$ .
12. À chaque étape, l'algorithme glouton consiste à faire un choix pour ajouter une case au chemin construit. Comme un chemin est de longueur  $2n - 1$ , il y a donc  $2n - 1$  étapes à effectuer (ou  $2n - 2$  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  $2n - 1$ , donc de l'ordre de  $n$  : son coût est linéaire.

### Partie C : Recherche par programmation dynamique

13. 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.
14. Voici le tableau complété :

s =

4	5	8	11
6	6	10	12
13	14	16	17
15	16	19	20

15. La case (0,0) de  $s$  est égale à  $\text{tab}[0][0]$ . 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  $\text{tab}$  pour obtenir la première colonne de  $s$ . De la même manière, il suffit de cumuler les valeurs de la première ligne de  $\text{tab}$  pour obtenir celles de  $s$ . Si on veut traduire ces constatations par des formules, cela donne :

- pour  $i$  différent de 0 :  $s[i][0] = s[i-1][0] + \text{tab}[i][0]$
- pour  $j$  différent de 0 :  $s[0][j] = s[0][j-1] + \text{tab}[0][j]$

16. 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  $\text{tab}[i][j]$ ) à cette somme et on obtient la somme maximale pour arriver en (i, j). C'est exactement la formule indiquée :

$$s[i][j] = \text{tab}[i][j] + \max(s[i-1][j], s[i][j-1])$$

17.

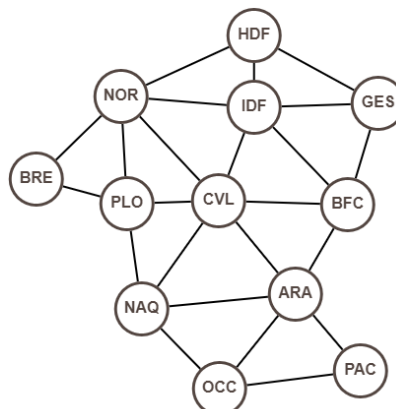
```
def somme_max_dynamique(tab):
    # création et initialisation du tableau
    n = len(tab)
    s = [[None] * n for i in range(n)]
    s[0][0] = tab[0][0]
    # remplissage du reste de la première ligne
    for j in range(1, n):
        s[0][j] = s[0][j-1] + tab[0][j]
    # remplissage du reste de 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]
```

18. L'algorithme remplit successivement les  $n \times n = n^2$  cases du tableau donc il y a  $n^2$  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 : 1 affectation ligne 5 (la case en haut à gauche) ;  $n - 1$  affectations ligne 8 (les  $n - 1$  autres cases de la première ligne) ;  $n - 1$  affectations ligne 11 (les  $n - 1$  autres cases de la première colonne) ;  $(n - 1) \times (n - 1) = (n - 1)^2$  affectations ligne 15 (les autres cases). En tout, on a donc  $1 + (n - 1) + (n - 1) + (n - 1)^2 = 2n - 1 + n^2 - 2n + 1 = n^2$  affectations.

## EXERCICE 2 (9 points)

### Partie A : Création d'un parcours de visites

1. Voici le graphe des régions :



2.

```
def voisins(G, s):
    return G[s]
```

3. 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.

4.

```
def sommets_a_distance(G, depart, n):
    visites = {depart: 0} # depart 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
```

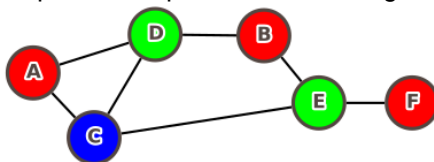
5. La recherche (d'une clé, ou d'une valeur) dans un dictionnaire se fait en temps constant alors que la recherche dans un tableau a un coût linéaire (dans le pire cas). Ainsi, tester si `voisin` est (ou n'est pas) dans `visites` est plus efficace si `visites` est un dictionnaire.

## Partie B : Coloration de la carte

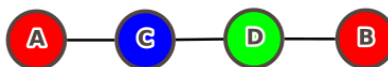
6. Les sommets sont coloriés dans l'ordre suivant :

- 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.

Sommet	Couleur
A	Rouge
B	Rouge
C	Bleu
D	Vert
E	Vert
F	Rouge



7. Voici le graphe G2 colorié :



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

8. 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
```

9. 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  $n^2$ , où  $n$  est la taille du tableau à trier). Le tri fusion est un exemple d'algorithme de tri plus efficace.

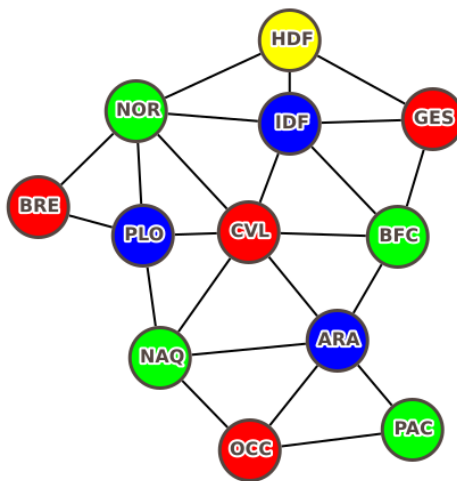
10. Voici la liste renvoyée contenant les sommets triés par degrés décroissants (le tri par insertion est stable, c'est-à-dire qu'il conserve l'ordre de départ en cas d'égalité) :
- ```
['CVL', 'ARA', 'IDF', 'NOR', 'PLO', 'NAQ', 'BFC', 'OCC', 'GES', 'HDF', 'BRE', 'PAC']
```
11. On peut par exemple, construire 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)

12. Il suffit de modifier la ligne 7 pour 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):`
13. 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

14. 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).
15. 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.
16. 

```
SELECT nom, latitude, longitude
FROM lieu
WHERE type = 'château';
```
17. 

```
SELECT nom
FROM lieu
WHERE type = 'musée' AND id_region = 1
ORDER BY nom;
```
18. 

```
UPDATE lieu
SET nom = 'Les Étoiles du Palais de la Découverte', latitude = 48.84113,
longitude = 2.27966
WHERE id = 6;
```
19. 

```
INSERT INTO lieu
VALUES (1253, 'La Géode', 'Paris', 48.894444, 2.388611, 'cinéma', 9);
```
20. 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';
```

21. 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.

22. 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

5. 11, 20, 32, 11, **20, 32, 11, 32, 11**
6. 11, 11, 20, 20, 32, 32, 11, 11, 32
7. 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

8. `liste_attente = [Processus(11, 4), Processus(20, 2), Processus(32, 3)]`
- 9.

```
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

```

10.

```

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

```