

Proposition de corrigé

Concours : Concours Commun INP

Année : 2023

Filière : PSI

Épreuve : Sciences Industrielles pour l'Ingénieur

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'[UPSTI](https://www.upsti.fr) (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

<https://www.upsti.fr/espace-etudiants/annales-de-concours>

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

Toute représentation ou reproduction (même partielle) de ce document faite sans l'accord de l'UPSTI est **interdite**. Seuls le téléchargement et la copie privée à usage personnel sont autorisés (protection au titre des [droits d'auteur](#)).

En cas de doute, n'hésitez pas à nous contacter à : corrigesconcours@upsti.fr.

Informez-vous !

Retrouvez plus d'information sur les [Sciences de l'Ingénieur](#), l'[orientation](#), les [Grandes Ecoles](#) ainsi que sur les [Olympiades de Sciences de l'Ingénieur](#) et sur les [Sciences de l'Ingénieur au Féminin](#) sur notre site : www.upsti.fr

L'équipe UPSTI

Reconnaissance optique de caractères

Corrigé UPSTI

Q1.

Il faut 8 bits par pixel. Un centimètre correspond à $300/2,5 = 120$ pixels environ.

Ainsi un format A4 correspond à une image de taille 2520×3560 pixels environ.

Il y a donc 9 000 000 pixels. Chaque pixel correspond à 24 bits (3 couleurs codées chacune sur 8 bits) ainsi l'image aura une taille de $216 \cdot 10^6$ bits.

Q2.

Pour chaque itération, on réalise 3 multiplications et 2 additions.

Ainsi on réalise au total : $5 \cdot p \cdot q$ opérations.

La complexité asymptotique est en $\Theta(p \cdot q)$, soit une complexité quadratique.

Q3.

def binarisation (imgG : array , seuil : int) -> array :

 p,q,_=dimension (imgG)

 img= initialise (p,q,0)

 for i in range (p) :

 for j in range (q) :

 if imgG [i] [j] > seuil :

 img [i] [j] = 255

 return img

Q4.

La réponse correcte est la fonction en haut à droite (1ere ligne, 2eme colonne).

Q5.

```
def rotation ( img : array , angle : float ) -> array :
```

```
    p , q , _ = dimension ( img )
```

```
    imr = initialise ( p , q , 255 )
```

```
    angr = -angle*3.14/180
```

```
    mat = [ [ cos ( angr ) , sin ( angr ) ] , [ - sin ( angr ) , cos ( angr ) ] ]
```

```
    for ni in range ( p ) :
```

```
        for nj in range ( q ) :
```

```
            x , y = prod_matrice_vecteur ( mat , [ ni - p // 2 , nj - q // 2 ] )
```

```
            x = x + p // 2
```

```
            y = y + q // 2
```

```
            if 0 <= x < p-1 and 0 <= y < q-1 :
```

```
                imr [ ni ] [ nj ] = bilineaire ( im , x , y )
```

```
    return int ( imr )
```

Q6.

Les pixels devant être entre 0 et 255 l'utilisation d'un type non signé permet de ne pas s'occuper du bit de signe.

Si on réalise $18 - 23 = 251$, le résultat négatif est non représentable en entier non signé. Le résultat de l'opération sera ainsi un nombre positif provoquant une erreur dans le calcul.

Q7.

```
def histo_ligne ( im : array ) -> list :
```

```
    h = [ ]
```

```
    p , q , _ = dimension ( im )
```

```
    for i in range ( p ) :
```

```
        s=0
```

```
        for j in range ( q ) :
```

```
            if im [ i ] [ j ] == 0 :
```

```
                s+=1
```

```
        h . append ( s )
```

```
    return h
```

Q8.

```
def detecter_lignes ( liste : list ) -> list :
```

```
    lignes = [ ]
```

```
    i = 0
```

```
    deb = -1
```

```
    while i < len ( liste ) :
```

```
        if liste [ i ] > 1 and deb == -1 :
```

```
            deb = i
```

```
        elif liste [ i ] < 1 and deb != -1 :
```

```
            lignes.append ( [ deb , i ] )
```

```
            deb = -1
```

```
        i +=1
```

```
    return lignes
```

Q9.

A chaque itération, la taille de l'intervalle est divisée par deux, donc au bout de n itérations, la taille de l'intervalle est $(b - a)/2^n$, qui tend vers 0 quand n augmente. Il existe ainsi un n_0 tel que la taille de l'intervalle est plus petite que ε : l'algorithme termine. La méthode utilisée est la dichotomie. On a $b_n - a_n = (b_{n-1} - a_{n-1}) / 2 = (b_0 - a_0) / 2^n < \varepsilon$ donc en inversant la relation, on obtient que

$n > \ln ((b_0 - a_0) / \varepsilon) / \ln 2$.

Q10.

```
def rotation_auto ( im : array , a : float , b : float ) -> array :
```

```
    c = ( a+b ) / 2
```

```
    f c = nb_zeros ( im , c )
```

```
    while b-a > 0.1 : # plus grand que 0.1 degré
```

```
        ac = ( a+c ) / 2
```

```
        fac = nb_zeros ( im , ac )
```

```
        cb = ( c+b ) / 2
```

```
        fcb = nb_zeros ( im , cb )
```

```
        maxi = max( fac , f c , fcb )
```

```
        if fac == maxi :
```

```
            b = c
```

```
            c = ac
```

fc = fac

elif fc == maxi :

a = ac

b = cb

else :

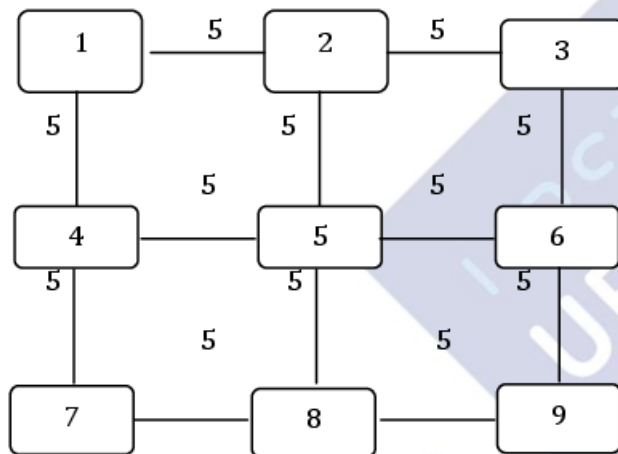
a = c

c = cb

fc = fcb

return rotation (im , (b+a) / 2)

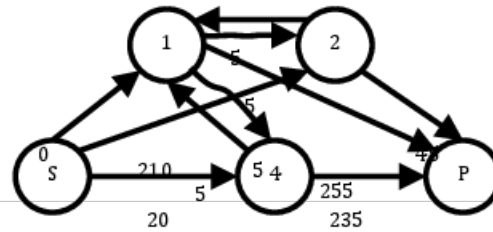
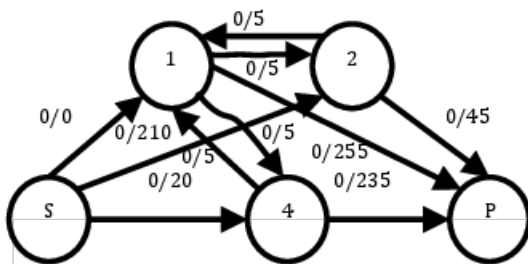
Q11.



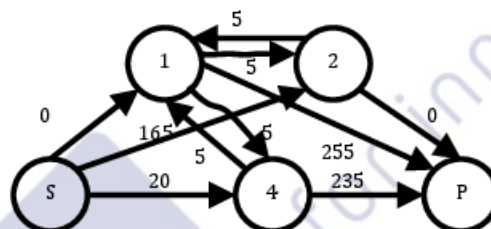
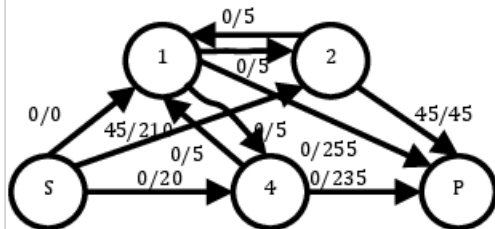
Q12.

	S	1	2	3	4	5	6	7	8	9	P
S	0	0	210	190	20	100	200	10	5	255	0
1		0	5	0	5	0	0	0	0	0	255
2			0	5	0	5	0	0	0	0	45
3				0	5	0	5	0	0	0	65
4					0	5	0	5	0	0	235
5						0	5	0	5	0	155
6							0	5	0	5	55
7								0	5	0	245
8									0	5	250
9										0	0
P											0

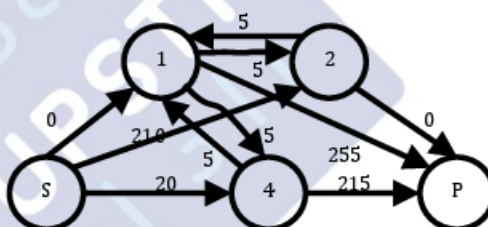
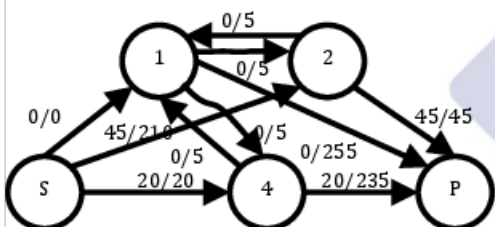
Q13.



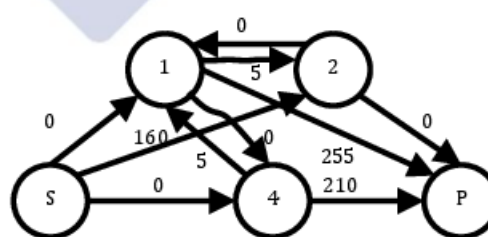
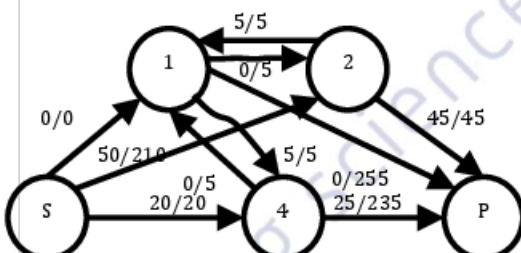
Chemin choisi : S 2 P. On augmente le flot de 45.



Chemin choisi : S 4 P. On augmente le flot de 20.



Chemin choisi : S 2 1 4 P. On augmente le flot de 5.



Chemin choisi : S 2 P. On augmente le flot de 45.

Il n'y a plus de chemins possibles car on peut aller de S vers 2 puis tous les chemins sont saturés.

Q14.

Le flot maximal obtenu est égal à $70 = 50 + 20$ obtenu en faisant la somme du flot sortant de S. Le seul partitionnement qui contient au moins un sommet en plus de S ou P correspond à S, 2 et 1, 4, P. La capacité de coupe a pour valeur $20 + 0 + 5 + 45 = 70$ ce qui correspond bien au flot maximal. On peut également noter que la somme des flots arrivant sur P vaut $70 = 45 + 0 + 25$.

Q15.

Les pixels noirs ne pourront jamais être reliés à la source car leur capacité est nulle. Les pixels du groupe A seront donc blancs et ceux du groupe B seront noirs. Les pixels gris (foncés) ont été pour la plupart regroupés automatiquement dans le groupe des pixels noirs. L'image obtenue correspond bien à la lettre t.

Q16.

```
SELECT id FROM FONTES
```

```
WHERE nom="Zurich" and style="romain" and taille between 10 and 16
```

Q17.

```
SELECT fichier FROM CARACTERES JOIN SYMBOLES ON SYMBOLES.id = id_symbole
```

```
WHERE label = "A"
```

Q18.

```
SELECT count(*) FROM CARACTERES
```

```
JOIN FONTES ON FONTES.id = id_fontes
```

```
JOIN SYMBOLES ON SYMBOLES.id = id_symbole
```

```
WHERE nom="Zurich" and style="romain" and taille between 10 and 16
```

```
GROUP BY label
```

Q19.

```
car = ["Zurich Light BT", "majuscules18", "10.png"] num="10"
```

```
var="majuscules" ind = 0 renvoie 'K'
```

Q20.

```
def lire_donnees_ref ( fichiers_car_ref : list ) -> dict :
```

```
    d = { }
```

```
    for f in fichiers_car_ref :
```

```
        symb = lire_symbole_fichier ( f )
```

```
        if symb in d :
```

```
            d [ symb ] . append ( imread ( f ) )
```

```
        else :
```

```
            d [ symb ] = [ imread ( f ) ]
```

```
    return d
```

Q21.

```
def distance ( im1 : array, im2 : array ) -> float :
```

```
    d = 0
```

```
    p, q, _ = dimension ( im1 )
```

```
    for i in range ( p ) :
```

```
        for j in range ( q ) :
```

```
            d = d + ( im1 [ i ] [ j ] - im2 [ i ] [ j ] ) **2
```

```
    return d **0.5
```

Q22.

```
def calcul_distances ( carac_ref: dict, carac_test : array ) -> dict:
```

```
    d = { }
```

```
    for symb in carac_ref:
```

```
        d [ symb ] = [ ]
```

```
        l = carac_ref [ symb ]
```

```
        for c in l:
```

```
            d [ symb ].append ( distance ( carac_test , c ) )
```

```
    return d
```

Q23.

Dans le pire des cas la méthode de tri fusion est performante et a une complexité en $\Theta(n \ln(n))$.

Q24.

```
def Kvoisins ( distances : dict, K : int ) -> list:
```

```
    voisins = [ ( float ( " inf " ) , "" ) for k in range (K) ]
```

```
    for lettre in distances:
```

```
        d = distances [ lettre ]
```

```
        for j in range ( len ( d ) ) :
```

```
            if d [ j ] < voisins [ - 1 ] [ 0 ] :
```

```
                k = len ( voisins ) - 1
```

```
            while k > 0 and voisins [ k - 1 ] [ 0 ] > d [ j ] :
```

```
                voisins [ k ] = voisins [ k - 1 ]
```

```
                k = k - 1
```

```
voisins [ k ] = [ d [ j ] , lettre ]
```

```
return voisins
```

Q25.

Les deux boucles lettre et j permettent de balayer les n éléments. Ensuite, on réalise au pire K fois les instructions. Ainsi la complexité est en $\Theta(n * K)$. Comme K est plus petit que $\log_2(n)$, on en déduit que cet algorithme est plus performant.

Q26.

```
def symbole_majoritaire ( voisins : list ) -> str :
```

```
    dico = { }
```

```
    for elem in voisins :
```

```
        if elem [ 1 ] in dico :
```

```
            dico [ elem [ 1 ] ] += 1
```

```
        else :
```

```
            dico [ elem [ 1 ] ] = 1
```

```
    maxi = 0
```

```
    symb = ""
```

```
    for key , num in dico.items ( ) :
```

```
        if num > maxi :
```

```
            maxi = num
```

```
            symb = key
```

```
    return symb
```

Q27.

On remarque que le nombre de voisins ne semble pas influencer la reconnaissance des caractères. Par contre on constate que plus on utilise des images des caractères plus la reconnaissance est correcte.