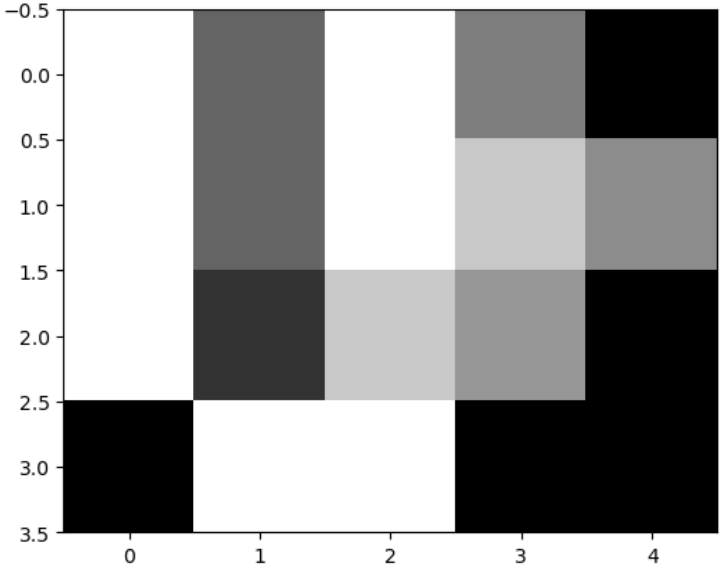
DM03 Images - Corrigé

In [1]:
```
from math import *
import numpy as np
import matplotlib.pyplot as plt
from copy import deepcopy
```

Images d'essai

Les cellules ci-dessous définissent et affichent 2 images. L'une, très petite, servira pour vérifier les algorithmes, l'autre servira comme exemple général.

In [4]:
```
M = [[255, 100, 255, 125, 0],
     [255, 100, 255, 200, 140],
     [255, 50, 200, 150, 0],
     [0, 255, 255, 0, 0]]

plt.imshow(M, cmap='gray')
plt.show() ; plt.close()
```



In [6]:
```
# Import d'une image et transformation en niveau de gris
from skimage.color import rgb2gray
img = plt.imread('647019.jpg') # import de l'image

img = rgb2gray(img) # conversion niveaux de gris : niveau = flottant compris ent
img = img * 255
img = [[round(img[i][j]) for j in range(len(img[0]))] for i in range(len(img))]
```

```
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.show(); plt.close()
```

Dupliquer une image

In [9]:
```
def copie(M):
    C = []
    for ligne in M:
        C.append(ligne[:])
    return C
```

??? tip Indice Si `L` est une liste d'éléments non modifiables (entiers, booléens, flottants, chaînes de caractères), alors `L[:]` réalise une copie des éléments de `L` . ???

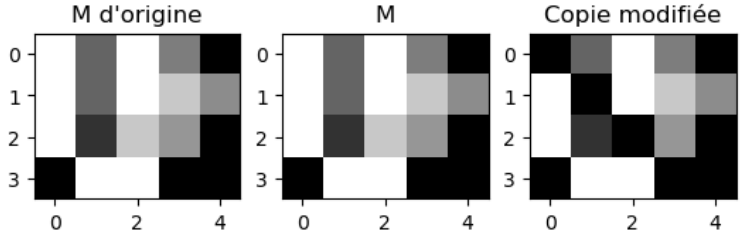
Q2

`L[:]` est de complexité linéaire, en p. D'ou une complexité en $\mathcal{O}(n \times p)$.

In [12]:
```
## Test de la fonction copie. Les redéfinitions de M permettent de récupérer la

M = [[255, 100, 255, 125, 0],
     [255, 100, 255, 200, 140],
     [255, 50, 200, 150, 0],
     [0, 255, 255, 0, 0]]

def testCopie(M):
    sauvegardeM = deepcopy(M)
    C = copie(M)  # test de la fonction copie
    for i in range(len(C)): # modification de la copie (M de doit pas être modif
        C[i][i] = 0
    print("M après modification de la copie\n", M)
    print("Copie après sa modification\n", C)

    OK = np.sum(np.array(sauvegardeM) - M) == 0
    if OK:
```

```
        print("\nTest réussit : la matrice d'origine n'a pas été modifiée lorsqu
    else:
        print("\nEchec du test : la modification de la copie de M a aussi modifi

    plt.subplot(131)
    plt.imshow(sauvegardeM, cmap='gray',vmin=0,vmax=255)
    plt.title("M d'origine")

    plt.subplot(132)
    plt.imshow(M, cmap='gray',vmin=0,vmax=255)
    plt.title("M")

    plt.subplot(133)
    plt.imshow(C, cmap='gray',vmin=0,vmax=255)
    plt.title("Copie modifiée")
    plt.show(); plt.close()

testCopie(M)

M = [[255, 100, 255, 125, 0],
     [255, 100, 255, 200, 140],
     [255, 50, 200, 150, 0],
     [0, 255, 255, 0, 0]]
```

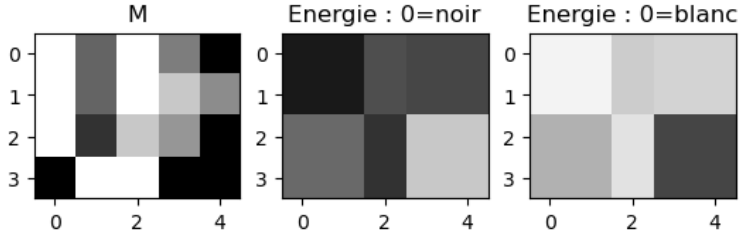
```
M après modification de la copie
 [[255, 100, 255, 125, 0], [255, 100, 255, 200, 140], [255, 50, 200, 150, 0], [0,
255, 255, 0, 0]]
Copie après sa modification
 [[0, 100, 255, 125, 0], [255, 0, 255, 200, 140], [255, 50, 0, 150, 0], [0, 255,
255, 0, 0]]

Test réussit : la matrice d'origine n'a pas été modifiée lorsque la copie a été m
odifiée. Les données sont dupliquées.
```



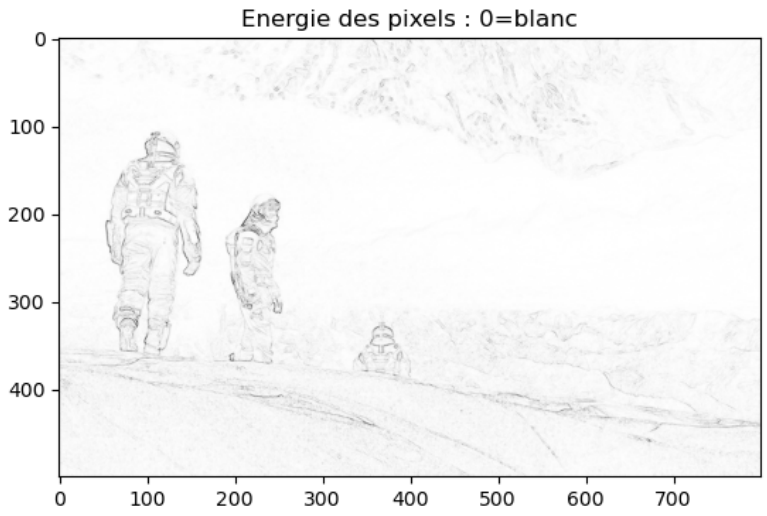
Energie des pixels : détection des contours

In [15]:
```
# Détection de contour d'une image en niveau de gris
def contours(T):
    n, p = len(T), len(T[0])
    contours = [None]
    for i in range(1, n-1): # pour toutes les lignes, sauf la première et la der
        ligne = [None]
        for j in range(1, p-1):
            val = abs(T[i+1][j] - T[i-1][j]) + abs(T[i][j+1] - T[i][j-1])
            val = round(val/2)
            ligne.append(val)
        ligne[0] = ligne[1]
        ligne.append(ligne[p-2])
        contours.append(ligne)

    contours[0] = contours[1][:]
    contours.append(contours[n-2][:])

    return contours


# code de test ____________________________________________
Me = contours(M)
print("Energie des pixels de M :")
print(Me)

plt.subplot(131)
plt.imshow(M, cmap='gray',vmin=0,vmax=255)
plt.title('M')

plt.subplot(132)
plt.imshow(Me, cmap='gray',vmin=0,vmax=255)
plt.title("Energie : 0=noir")

plt.subplot(133)
plt.imshow(Me, cmap='Greys',vmin=0,vmax=255)
plt.title("Energie : 0=blanc")
plt.show(); plt.close()
```

```
Energie des pixels de M :
[[25, 25, 78, 70, 70], [25, 25, 78, 70, 70], [105, 105, 50, 200, 200], [105, 105,
50, 200, 200]]
```



In [17]:
```
tmp = deepcopy(img)
plt.imshow(contours(img), cmap='Greys')
plt.title("Energie des pixels : 0=blanc")
plt.show(); plt.close()
```



Q4

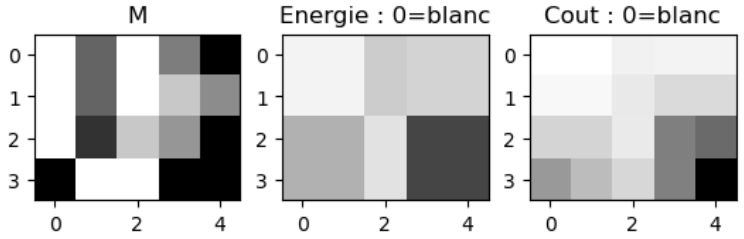
La fonction comprend 2 boucles de n et p passages. Un passage à i et j donnés est à cout constant. Les boucles sont donc de complexité $\mathcal{O}(n \times p)$. Les boucles sont suivies des copies de la première et dernière lignes, de complexité $\mathcal{O}(p)$.

La complexité de la fonction est donc $\mathcal{O}(n \times p)$.

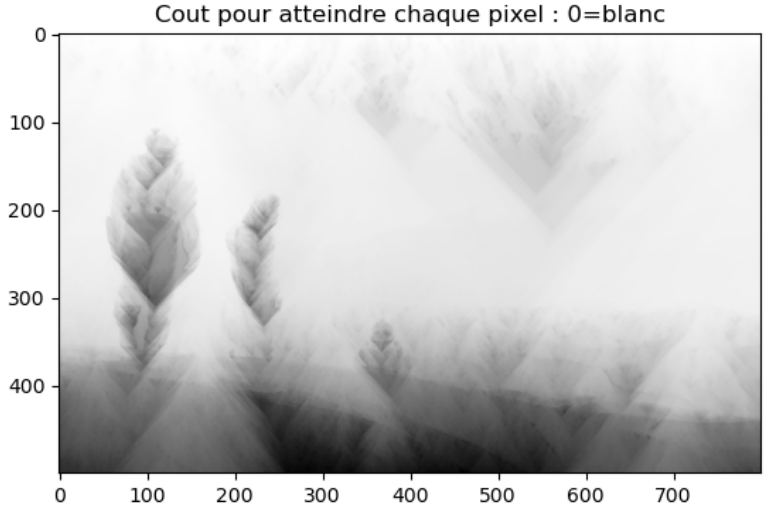
Cout des chemins

In [20]:
```python
def cout(T):
    n, p = len(T), len(T[0])
    C = contours(T)
    for i in range(1, n): # pour chaque ligne sauf la première
        C[i][0] = C[i][0] + min(C[i-1][0], C[i-1][1]) # premier pixel
        C[i][-1] = C[i][-1] + min(C[i-1][-1], C[i-1][-2]) # dernier pixel
        for j in range(1, p-1): # pour les autres pixels de la ligne
            C[i][j] = C[i][j] + min(C[i-1][j-1], C[i-1][j], C[i-1][j+1])
    return C


# code de test ____________________________________________________
Me = contours(M)
Mc = cout(M)

print("Cout pour atteindre un pixel de M :")
print(Mc)

plt.subplot(131)
plt.imshow(M, cmap='gray',vmin=0,vmax=255)
plt.title('M')

plt.subplot(132)
plt.imshow(Me, cmap='Greys',vmin=0,vmax=255)
plt.title("Energie : 0=blanc")

plt.subplot(133)
plt.imshow(Mc, cmap='Greys')
plt.title("Cout : 0=blanc")
plt.show(); plt.close()
```

```
Cout pour atteindre un pixel de M :
[[25, 25, 78, 70, 70], [50, 50, 103, 140, 140], [155, 155, 100, 303, 340], [260,
205, 150, 300, 503]]
```



In [22]:
```python
tmp = deepcopy(img)
plt.imshow(cout(img), cmap='Greys')
plt.title("Cout pour atteindre chaque pixel : 0=blanc")
plt.show() ; plt.close()
```



Q6

La fonction comprend 2 boucles de n et $p-1$ passages. Les lignes 5, 6 et 8 sont à cout constant. Un passage à i et j donnés est à cout constant. Les boucles sont donc de complexité $\mathcal{O}(n \times p)$. Les boucles sont précédés, ligne 3, d'une fonction de complexité identique.

La complexité de la fonction est donc $\mathcal{O}(n \times p)$.

Recherche d'un chemin

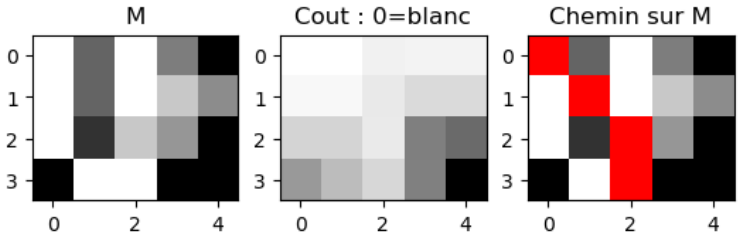
In [25]:
```
def imini(L):
    imin = 0
    vmin = L[0]
    for i in range(1,len(L)):
        v = L[i]
        if v < vmin:
            imin = i
            vmin = v
    return imin

# test
imini(M[1])==1
```

Out[25]: True

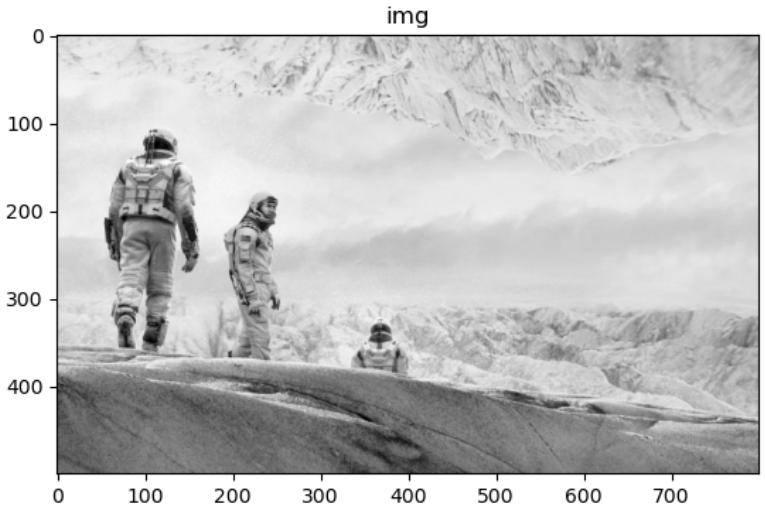
In [27]:
```
def chemin(T):
    n, p = len(T), len(T[0])

    C = cout(T)
    chemin = [0] * n # initialisation du chemin
    col = imini(C[-1]) # début du chemin sur la dernière ligne
    chemin[-1] = col

    i = n - 2 # à partir de l'avant dernière ligne
    while i >= 0:
        b = C[i][col]
        if col > 0 :
            a = C[i][col-1]
        else:
            a = b + 1
        if col < p - 1:
            c = C[i][col+1]
        else:
            c = b + 1

        if a<=b and a<=c: # mini en a, 1 à gauche
            col = col-1
        elif c<b and c<a:# mini en c, 1 à droite
            col = col + 1
        chemin[i] = col
        i = i - 1
    return chemin

# test
chemin(M) == [0, 1, 2, 2]
```

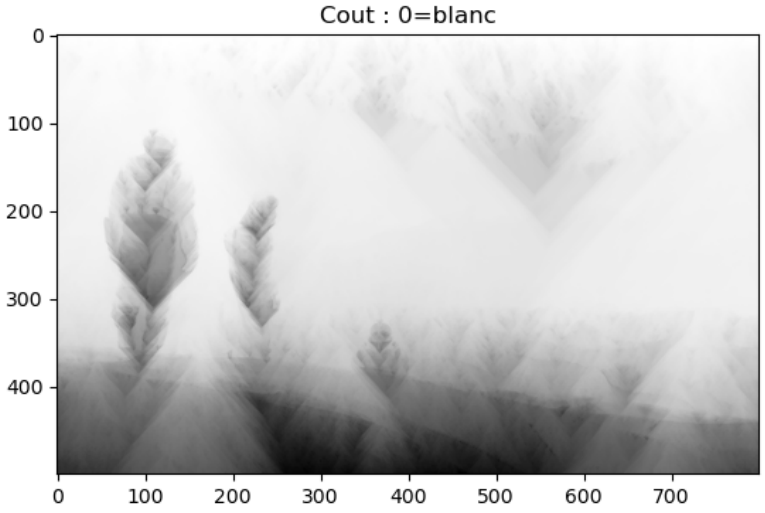
Out[27]: True

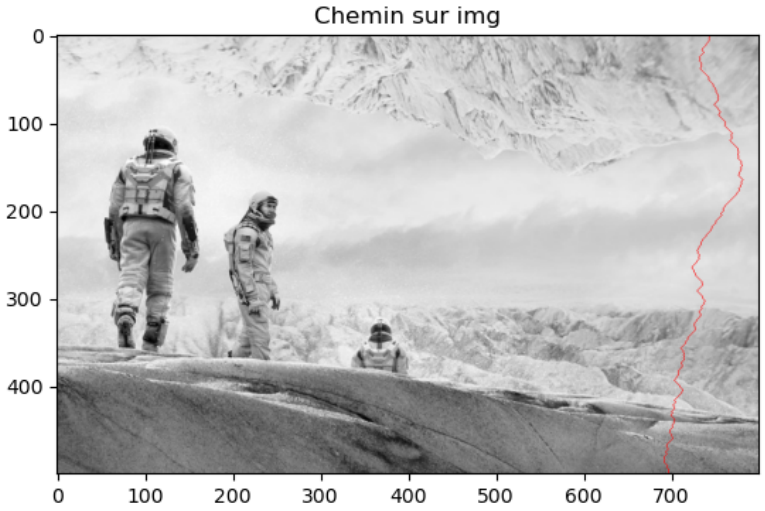
In [29]:
```
## Fonctions de test : copy l'image, l'affiche, ainsi que le cout et le chemin s
from copy import deepcopy
from skimage.color import gray2rgb
def traceChemin(img, fic=None):
    '''Affiche l'image img avec le chemin en superposition.
    Si un nom de fichier est donné (une chaîne de caractères), 'image.png' par e
    img = deepcopy(img)
    c = chemin(img)
    img = gray2rgb(img)
    maxi = np.max(img)
    for i in range(len(img)):
        img[i][c[i]] = (maxi,0,0)

    plt.imshow(img)
    if fic is not None:
        plt.imsave(fic, img, cmap='gray')


plt.subplot(131)
plt.imshow(M, cmap='gray',vmin=0,vmax=255)
plt.title('M')

plt.subplot(132)
plt.imshow(cout(M), cmap='Greys')
plt.title("Cout : 0=blanc")

plt.subplot(133)
traceChemin(M)
plt.title("Chemin sur M")
plt.show() ; plt.close()
```



In [31]:
```
## Affichage des résutlats sur l'image test
plt.figure(1)
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.title('img')
plt.show()

plt.figure(2)
plt.imshow(cout(img), cmap='Greys')
plt.title("Cout : 0=blanc")
plt.show()

plt.figure(3)
traceChemin(img)
plt.title("Chemin sur img")
plt.show() ; plt.close('all')
```

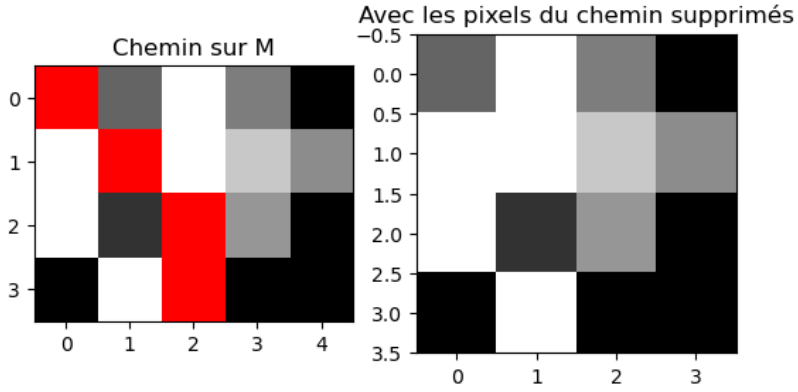






Réduction de l'image

In [34]:
```
def reduire1(T):
    c = chemin(T)
    for i in range(len(T)):
        T[i].pop(c[i])
    return T


# code de test
plt.subplot(121)
traceChemin(M)
plt.title("Chemin sur M")

plt.subplot(122)
tmp = deepcopy(M) # permet de conserver l'image d'origine
_ = plt.imshow(reduire1(tmp), cmap='gray', vmin=0, vmax=255)
plt.title("Avec les pixels du chemin supprimés")
plt.show() ; plt.close()

print("Matrice M réduite :")
print(tmp)
```




```
Matrice M réduite :
[[100, 255, 125, 0], [255, 255, 200, 140], [255, 50, 150, 0], [0, 255, 0, 0]]
```

In [36]:
```
tmp = deepcopy(img)
print("Taille image avec réduction :", len(img), "x", len(img[0]))
tmp = reduire1(tmp)
plt.imshow(tmp, cmap='gray', vmin=0, vmax=255)
plt.show() ; plt.close()
print("Taille image après réduction :", len(tmp), "x", len(tmp[0]))
```

```
Taille image avec réduction : 500 x 800
```

```
Taille image après réduction : 500 x 799
```

In [38]:
```
def reduire(T, k):
    T = deepcopy(T)
    for i in range(k):
        T = reduire1(T)
    return T

tmp = reduire(img,130) # réduction de 130 pixels de large
plt.imshow(tmp, cmap='gray', vmin=0, vmax=255)
plt.show() ; plt.close()
print("Taille image après réduction :", len(tmp), "x", len(tmp[0]))
```

```
Taille image après réduction : 500 x 670
```

Q11

La fonction appelle k fois la fonction `reduire1`.

Cette fonction appelle la fonction `chemin`. `chemin` appelle la fonction `cout` de complexité $\mathcal{O}(n \times p)$ suivie d'une boucle de n passages à cout constant. `chemin` a donc la même complexité que `cout`.

La complexité de la fonction est donc $\mathcal{O}(n \times p \times k)$.