lezione12

November 2, 2023

1 Fondamenti di Programmazione

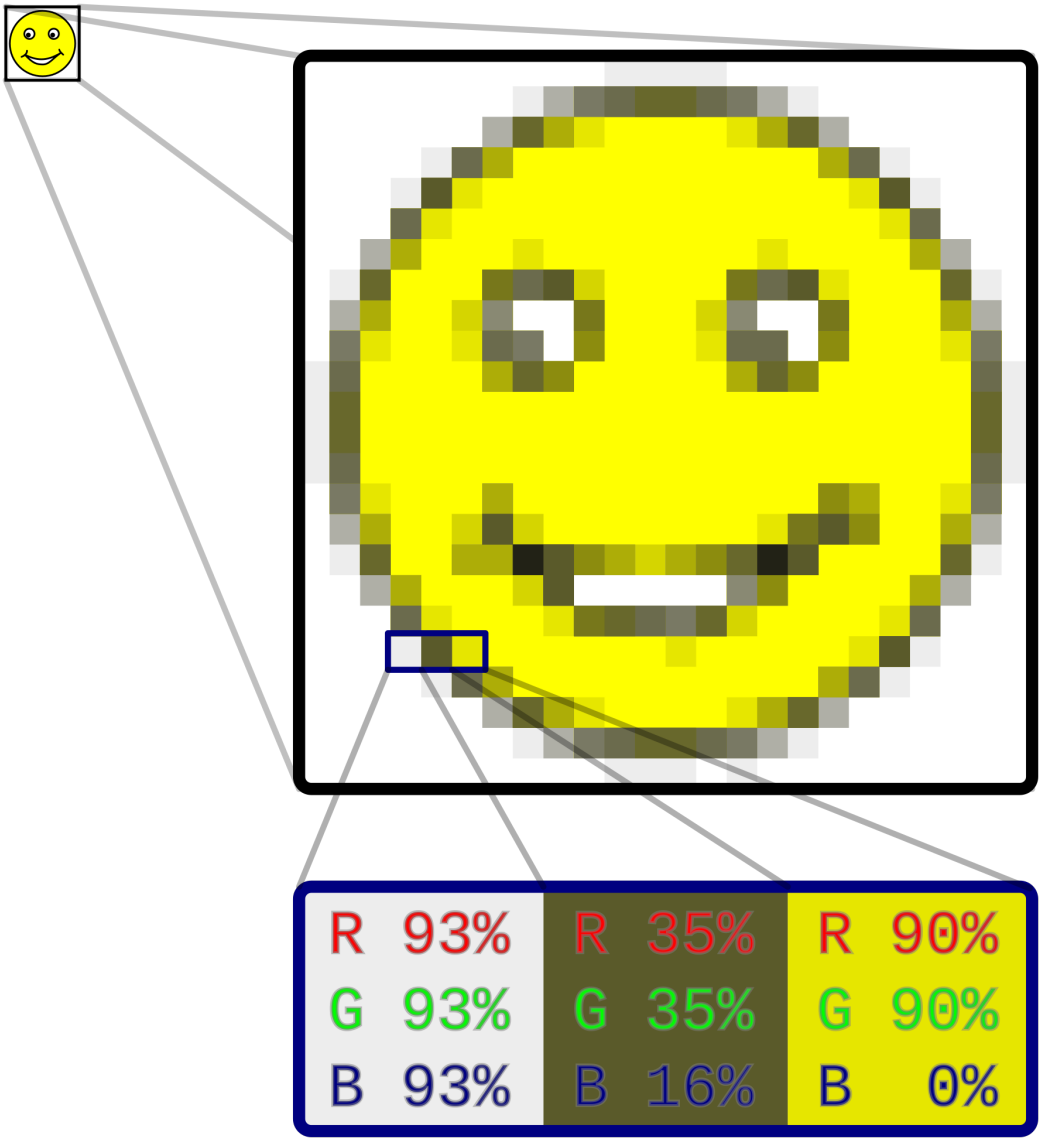
Andrea Sterbini

lezione 12 - 2 novembre 2023

2 RECAP:

- **query in un gruppo di files con tf-idf (term frequency - inverse document frequency)**
- files **json**
- files **yaml**
- lettura di pagine e file da web con **requests**

3 OGGI: immagini



- sono griglie rettangolari di pixel (**pic**ture **el**ement) colorati
- ogni posizione a coordinate x,y contiene un colore
- **RGB** è un modo di codificare i colori (altri Color spaces: HSV/L/B, CMYK)
  - R: luminosità della componente rossa (red)
  - G: luminosità della componente verde (green)
  - B: luminosità della componente blu (blue)

Questi valori in genere sono codificati in un byte ciascuno [0 .. 255] quindi un pixel occupa 3*8=24 bit (16M colori)

[1]:
```
%load_ext nb_mypy
```

Version 1.0.5

[2]:
```
Pixel   = tuple[int,int,int]
# con valori tra 0 e 255 compresi (1 byte)
# definiamo qualche colore
black : Pixel =   0,   0,   0    # luminosità minime
white : Pixel = 255, 255, 255    # luminosità massime
red   : Pixel = 255,   0,   0
green : Pixel =   0, 255,   0
blue  : Pixel =   0,   0, 255
cyan  : Pixel =   0, 255, 255
yellow: Pixel = 255, 255,   0
purple: Pixel = 255,   0, 255
grey  : Pixel = 128, 128, 128
```

3.1 Immagine = matrice di pixel = lista di liste di triple

- per semplicità usiamo una **lista di liste**
- la **lista esterna** è l'immagine, che contiene **righe orizzontali di pixel**
- ciascuna **riga** di pixel è una **lista di pixel**
- ciascun **pixel** è rappresentato da una terna **(R, G, B)** con valori interi tra [0..255]
- tutte le righe hanno la stessa lunghezza
- le righe nella immagine sono disposte dall'alto in basso (l'asse Y va in giù)
- ciascuna riga va da sinistra a destra (l'asse X va verso destra)

NOTA: Le immagini definite così sono una scusa per farvi manipolare matrici. Per manipolare immagini davvero si usa la libreria **Pillow** che non useremo nel corso.

[3]:
```
import images
# esempio di bandiera    rossa bianca verde
img = [
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
```

```
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
        [(255,0,0), (255,0,0), (255,0,0), (255,255,255), (255,255,255),␣
 ↪(255,255,255), (0,255,0), (0,255,0), (0,255,0) ],
      ]

images.visd(img)
```

3.2 Come trovare un pixel

Se **img** contiene la lista di righe

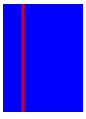
img[y] è la riga **y** esima (partendo dall'alto)

img[y][x] è il pixel della riga **y** che si trova a colonna **x** (da sinistra a destra)

4 Creiamo di una immagine/matrice monocolore !!! MA nel modo sbagliato !!!

[4]:
```
Line    = list[Pixel]
Picture = list[Line]

def crea_immagine_errata(larghezza : int, altezza : int, colore : Pixel) ->␣
 ↪Picture :
    riga = [ colore ] * larghezza     # creo una lista di pixel
    img  = [ riga   ] * altezza       # creo una lista di righe
    return img
```

[5]:
```
img2 : Picture = crea_immagine_errata(30, 40, blue)

img2[5][7] = red     # Provo a colorare un solo pixel

images.visd(img2)    # e trovo una colonna rossa!!!
```



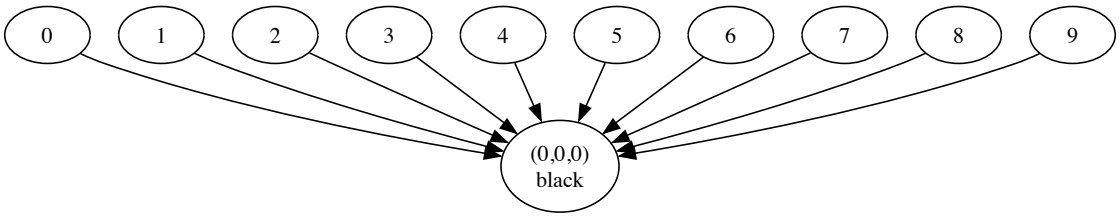
4.1 PERCHE' viene una riga verticale invece che un punto?

L'istruzione **`riga = [ colore ] * larghezza`** costruisce una lista di RIFERIMENTI ad un **unico** colore in memoria

Ciascuna posizione della lista/riga indica la stessa unica tripla RGB

[6]:
```
# figura di una lista di riferimenti allo stesso colore
from pygraphviz import AGraph
G = AGraph(rankdir='TD', directed=True)
for N in range(10):
    if N:
        G.add_edge(N-1, N, size=0, color='white')
    G.add_edge(N, "(0,0,0)\nblack")
G.add_subgraph(list(range(10)), rank="same")
G.layout('dot')
G
```

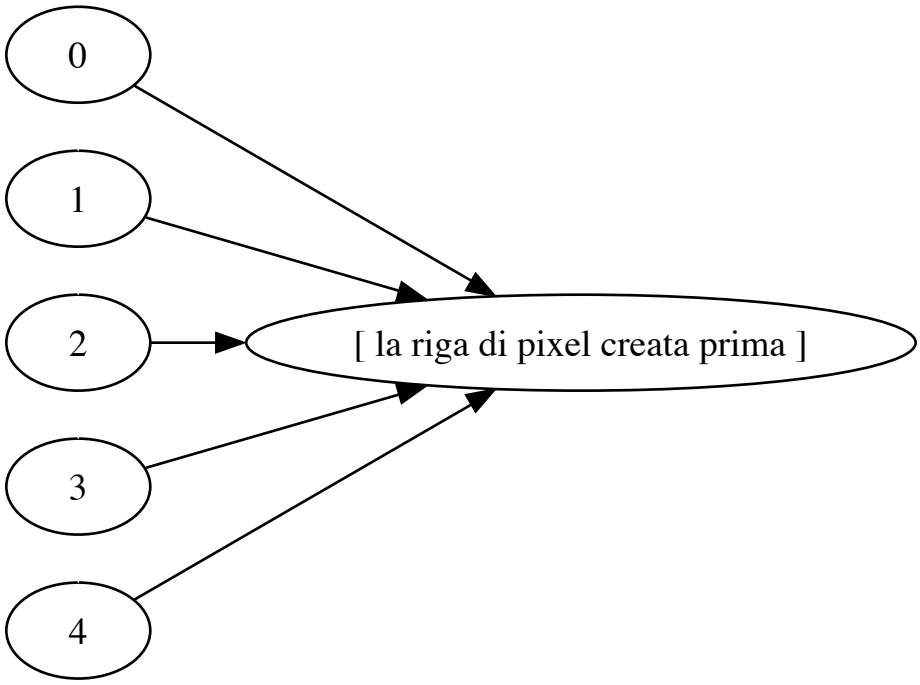
[6]:



L'istruzione **`img = [ riga ] * altezza`** costruisce una lista di RIFERIMENTI ad una unica riga in memoria

[7]:
```
# figura di una lista di riferimenti alla stessa lista
G = AGraph(rankdir='LR', directed=True)
for N in range(5):
    if N:
        G.add_edge(N-1, N, size=0, color='white')
    G.add_edge(N, "[ la riga di pixel creata prima ]")
G.add_subgraph(list(range(5)), rank="same")
G.layout('dot')
G
```

[7]:



Mentre **la prima istruzione va bene** perchè tutti i colori sono tuple, **immutabili**

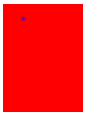
La seconda NON VA BENE perchè vogliamo che le righe siano **modificabili indipendentemente** l'una dall'altra

4.2 Creiamo l'immagine nel modo giusto

[8]:
```
def crea_immagine(larghezza : int, altezza : int, colore : Pixel=black) ->␣
↪Picture:
    "costruzione di una immagine con una list-comprehension"
    return [  [ colore ]*larghezza            # creo una nuova riga di pixel␣
↪'colore'
                    for _ in range(altezza)   # e lo faccio 'altezza' volte␣
↪indipendentemente
            ]

def crea_imm(larghezza : int, altezza : int, colore: Pixel=black) -> Picture:
    "qui faccio lo stesso ma senza list-comprehension"
    img = []                            # immagine = lista di righe inizialmente␣
↪vuota
    for y in range(altezza):            # per altezza volte
        riga = []                       # creo una riga = lista di pixel␣
↪inizialmente vuota
        for x in range(larghezza):      # e per larghezza volte
            riga.append(colore)         # aggiungo il pixel alla riga corrente
```

```
        img.append(riga)                # e poi aggiungo la riga all'immagine
    return img                          # torno la lista di liste finale
```

[9]:
```
img = crea_immagine(30, 40, red)
img[5][7] = blue      # coloro un pixel

images.visd(img)      # e ora va molto meglio, si colora un solo punto
```



5 Come caricare/salvare una immagine da/su disco

[10]:
```
# images.load(filename : str) -> Picture

img3 = images.load('3cime.png')

print(len(img3), len(img3[0]))    # altezza e larghezza
images.visd(img3)                 # visualizza la Picture in Jupyter
```

```
183 275
```

[11]:
```
img3[40][30:250] = [red]*220      # coloriamo una fila orizzontale di pixel␣
↪usando le slice
                                  # ATTENZIONE: il numero di pixel deve essere␣
↪quello giusto
```

```
# images.save(img : Picture, filename : str) -> None
images.save(img3, '3cime-2.png')

images.visd(img3)
```

6 Disegnare un pixel a coordinate qualsiasi

MA ... senza generare errori se le coordinate sono fuori dall'immagine

[12]:
```
# Opzione 1 --- controllando le posizioni
def draw_pixel(img : Picture, x : int|float, y : int|float, colore : Pixel):
    # ricavo l'altezza e larghezza dell'immagine contando righe e colonne
    A,L = len(img), len(img[0])
    x = int(round(x))       # voglio gestire anche coordinate float
    y = int(round(y))
    # cambio il pixel solo se è dentro l'immagine
    if 0 <= x < L and 0 <= y < A:
        img[y][x] = colore       # nella riga y e nella colonna x metto il␣
 ↪colore
```

[13]:
```
# Riprendo l'esempio
draw_pixel(img3, 20, 20, red)

images.visd(img3)
```

7 Gestione degli errori con Try/except/finally

Per catturare eventuali errori e gestirli nel proprio programma si usa

```
try:
    codice che potrebbe produrre un errore
except TipoDiErrore as e:
    codice da eseguire se TipoDiErrore
except AltroTipo:
    ...
finally:
    codice da eseguire alla fine SEMPRE
```

NOTA: la clausola **except:** che da sola cattura TUTTE le eccezioni **E' FORTEMENTE SCONSIGLIATA** perchè potrebbe nascondere degli errori che non vi aspettate e che vanno gestiti diversamente

[14]:
```
# Secondo modo: --- usando try-except per catturare l'errore di sbordamento?
def draw_pixel_wrong(img : Picture, x : int, y : int, colore : Pixel):
    # mi preparo a catturare l'errore (try)
    try:
        img[y][x] = colore          # provo a disegnare il pixel a coordinate x,y
    except IndexError:
        pass                        # se c'è errore di index nelle liste lo␣
 ↪ignoro

# --- BEWARE of negative indexes!!! (che non producono errori ma lavorano a␣
 ↪ritroso nelle liste)

# --- BEWARE of generic 'catch-all' except clauses!!!!! (che nascondono TROPPI␣
 ↪errori)
```

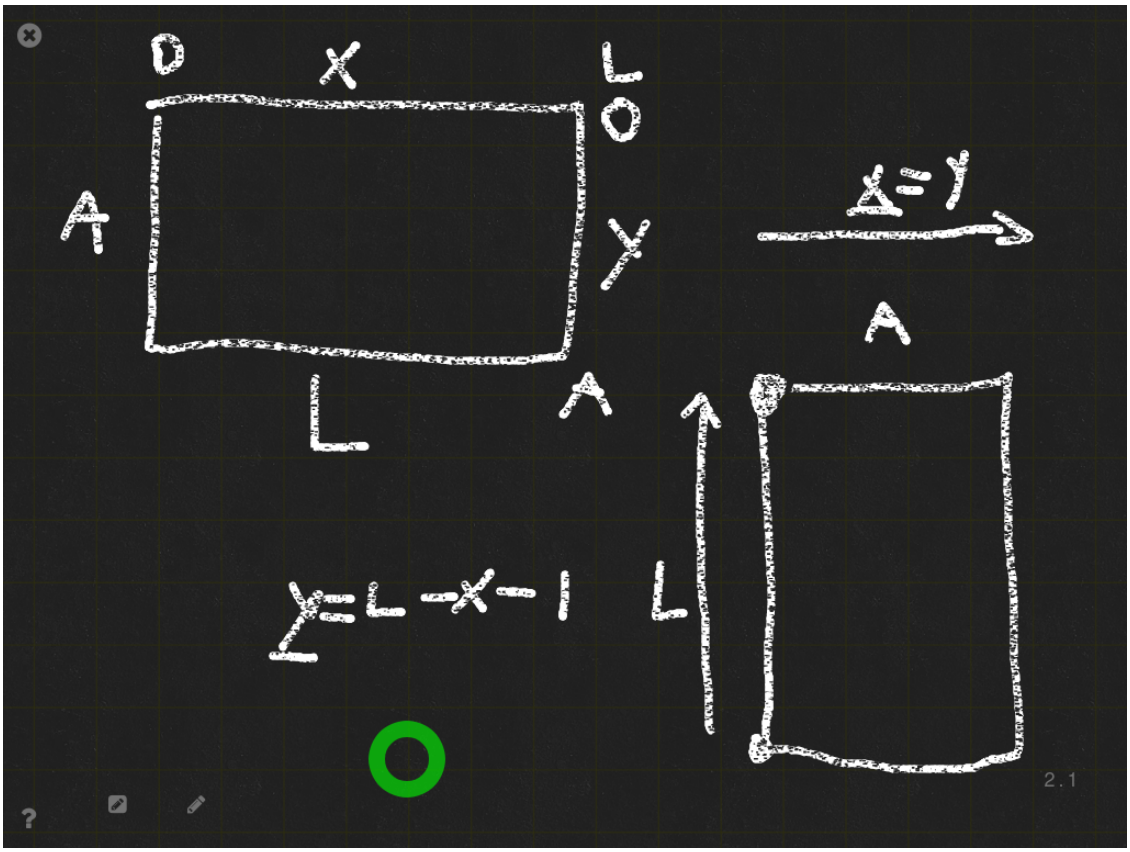
```
for i in range(-1000,0):
    draw_pixel_wrong(img3, i, i, green)   # disegno fuori in alto a sinistra
images.visd(img3)
```

[15]:
```
def draw_pixel_maybe_better(img : Picture, x : int|float, y : int|float, colore
 ↪: Pixel):
    x = int(round(x))       # voglio gestire anche coordinate float
    y = int(round(y))
    A,L = len(img), len(img[0])
    # mi preparo a catturare l'errore (try)
    try:
        # controllo le coordinate e lancio un errore se sbordo
        assert 0 <= x < L and 0 <= y < A , f"coordinate FUORI {x},{y}"
        img[y][x] = colore          # disegno il pixel
    except AssertionError as e:     # se l'asserzione era falsa ho sbordato
 ↪dall'immagine
        print(e)                    # stampo l'eccezione
        pass

# MA questo è lo stesso che usare un if!!! NON NE VALE LA PENA
for i in range(-20,0):
    draw_pixel_maybe_better(img3, i, i, red)

images.visd(img3)
```

```
coordinate FUORI -20,-20
coordinate FUORI -19,-19
coordinate FUORI -18,-18
coordinate FUORI -17,-17
coordinate FUORI -16,-16
```

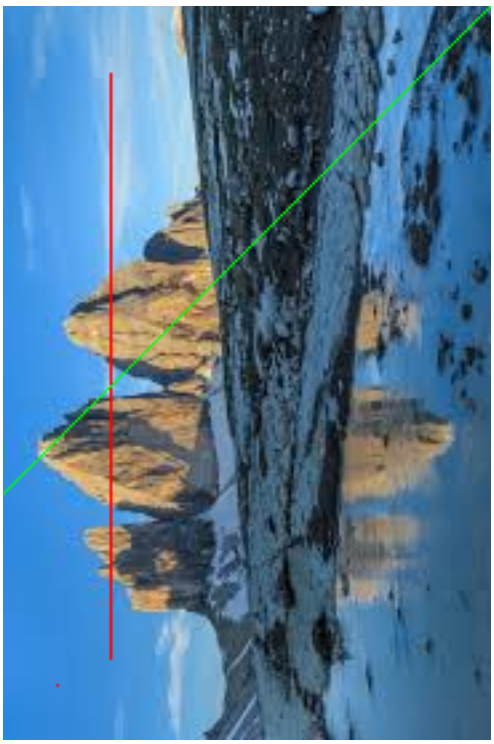
```
coordinate FUORI -15,-15
coordinate FUORI -14,-14
coordinate FUORI -13,-13
coordinate FUORI -12,-12
coordinate FUORI -11,-11
coordinate FUORI -10,-10
coordinate FUORI -9,-9
coordinate FUORI -8,-8
coordinate FUORI -7,-7
coordinate FUORI -6,-6
coordinate FUORI -5,-5
coordinate FUORI -4,-4
coordinate FUORI -3,-3
coordinate FUORI -2,-2
coordinate FUORI -1,-1
```

8 Rotazione di 90° a sinistra (antioraria)



[16]:
```
# X_destinazione = y_sorgente
# Y_destinazione = larghezza_sorgente - 1 - x_sorgente

def ruota_sx(img):
    altezza, larghezza = len(img),len(img[0])      # ottengo le dimensioni
    img2 = crea_immagine(altezza, larghezza)       # creo una immagine nera con␣
 ↪altezza e larghezza scambiate
    for y, riga in enumerate(img):                 # per ogni pixel della␣
 ↪immagine originale
        for x, pixel in enumerate(riga):
            X = y                                  # calcolo le coordinate X,Y␣
 ↪della destinazione
            Y = larghezza -1 -x
            img2[Y][X] = pixel                     # e copio il pixel nella␣
 ↪destinazione
    return img2                                    # torno l'immagine ruotata

img_r = ruota_sx(img3)

images.visd(img_r)
```

```
# --- PER CASA: rotazione destra
```



9 Disegnare una linea orizzontale o verticale

[105]:
```
def draw_h_line(img, x, y, x2, colore):
    altezza   = len(img)
    if 0 <= y < altezza:                    # SOLO se la y è tra 0 e altezza
        if x>x2 : x, x2 = x2, x             # scambio i valori se non sono␣
 ↪nell'ordine giusto
        for X in range(x, x2+1):            # scandisco le X da x a x2
            draw_pixel(img, X, y, colore)  # riusiamo la draw_pixel che␣
 ↪controlla di non sbordare

# oppure prima intersechiamo la linea con l'immagine e poi la disegnamo senza␣
 ↪controllare
def draw_h_line2(img, x, y, x2, colore):
    altezza   = len(img)
    if 0 <= y < altezza:                    # SOLO se la y è tra 0 e altezza
        if x>x2 : x, x2 = x2, x             # scambio i valori se non sono␣
 ↪nell'ordine giusto
        larghezza = len(img[0])
```

```
        # la parte da disegnare ha estremi non maggiori di larghezza-1 e non␣
 ↪minori di 0
        xmin = min(max(x, 0), larghezza-1)
        xmax = max(min(x2, larghezza-1),0)
        # una volta aggiustati gli estremi la si disegna SENZA CONTROLLI!
        img[y][xmin:xmax+1] = [colore]*(xmax-xmin+1)    # con un assegnamento a␣
 ↪slice

img = images.load('3cime.png')
%time draw_h_line( img, 300, 100, 200, red)    # x>x2
%time draw_h_line2(img, 300, 150, 200, red)    # x>x2       questa è più rapida

images.visd(img)
```

```
CPU times: user 43 µs, sys: 1e+03 ns, total: 44 µs
Wall time: 44.8 µs
CPU times: user 6 µs, sys: 0 ns, total: 6 µs
Wall time: 6.91 µs
```

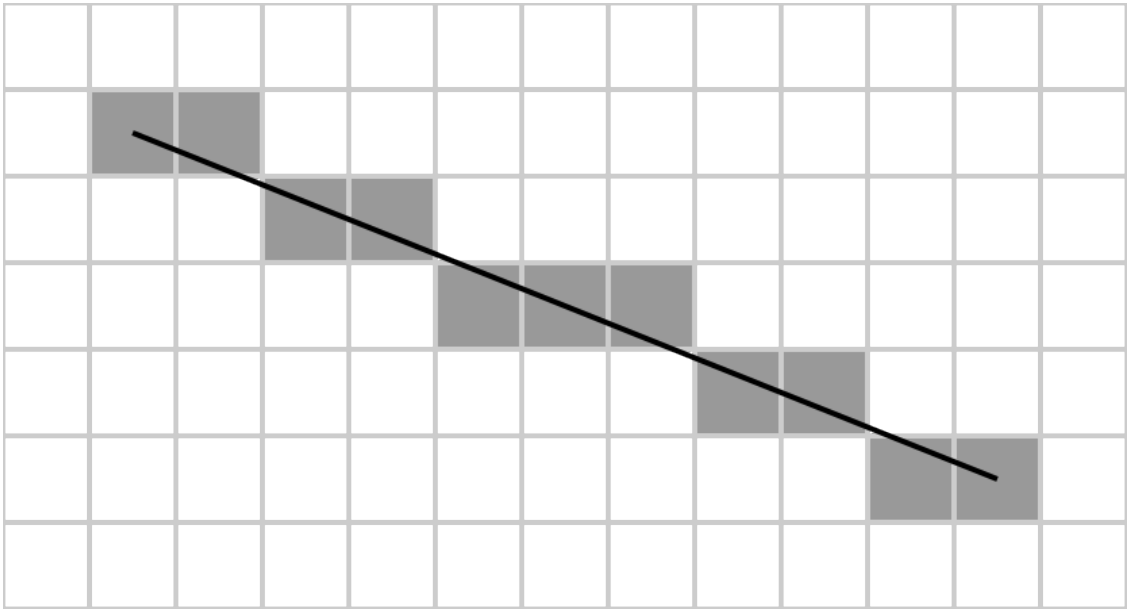
[106]:
```
# lo stesso per una linea verticale
def draw_v_line(img, x, y, y2, colore):
    larghezza = len(img[0])
    if 0 <= x < larghezza:                # SOLO se la x è tra 0 e larghezza
        if y>y2 : y, y2 = y2, y           # scambio i valori se non sono␣
 ↪nell'ordine giusto
        for Y in range(y, y2+1):
            # riusiamo la draw_pixel che controlla di non sbordare
            draw_pixel(img, x, Y, colore)
```

```
# oppure prima intersechiamo la linea con l'immagine e poi la disegnamo senza␣
 ↪controllare
def draw_v_line2(img, x, y, y2, colore):
    larghezza = len(img[0])
    if 0 <= x < larghezza:                # SOLO se la x è tra 0 e larghezza
        if y>y2 : y, y2 = y2, y           # scambio i valori se non sono␣
 ↪nell'ordine giusto
        altezza   = len(img)
        # la parte da disegnare ha estremi non maggiori di altezza-1 e non␣
 ↪minori di 0
        ymin = min(max(y, 0), altezza-1)
        ymax = max(min(y2, altezza-1),0)
        # una volta aggiustati gli estremi la si disegna SENZA CONTROLLI!
        for Y in range(ymin, ymax+1):     # qui è necessario fare un ciclo
            img[Y][x] = colore
```

[19]:
```
img = images.load('3cime.png')
%time draw_v_line( img, 50, 100, 200, red)
%time draw_v_line2(img, 30, 100, 200, red)       # più rapida

images.visd(img)
```

```
CPU times: user 43 µs, sys: 0 ns, total: 43 µs
Wall time: 45.1 µs
CPU times: user 12 µs, sys: 1 µs, total: 13 µs
Wall time: 13.1 µs
```

10 E per le diagonali?



Conviene scandire il cateto **più lungo** in modo che non ci siano colonne/righe senza pixel

[20]:
```
# --- e diagonale??? come???
# dipende dalla direzione

def draw_slope(img, x1, y1, x2, y2, colore):
    dx = x2-x1
    dy = y2-y1

    # ci si sposta lungo la direzione più lunga
    # se |dx| > |dy| si calcola la y per ciascuna x in [x1 .. x2]
    if abs(dx) >= abs(dy):
        if x1 > x2 : x1, x2 = x2, x1             # mi assicuro che x1<x2
        m = dy/dx                                # calcolo il coefficiente␣
 ↪angolare m=dy/dx
        for x in range(x1, x2+1):                # e per ogni x in [x1 .. x2]
            y = m * (x - x1) + y1                # calcolo y = mx+c      con␣
 ↪c=y1
            draw_pixel(img, x, y, colore)        # e disegno il pixel (con␣
 ↪draw_pixel?)
    # altrimenti per ciascuna y calcoliamo la x
    else:            # semplicemente scambiando x e y nelle formule precedenti
        if y1 > y2 : y1, y2 = y2, y1             # mi assicuro che y1<y2
        m = dx/dy                                # calcolo m=dx/dy
        for y in range(y1, y2+1):                # e per ogni y in [y1 .. y2]
            x = m * (y - y1) + x1                # calcolo x = my+c      con␣
 ↪c=y1
```

```
            draw_pixel(img, x, y, colore)        # e disegno il pixel (con␣
↪draw_pixel?)
```

[21]:
```
img = images.load('3cime.png')
%time draw_slope(img, 10,20, 150, 200, green)
%time draw_slope(img, 10,20, 10, 200, green)
%time draw_slope(img, 10,20, 150, 20, green)
images.visd(img)
```

```
CPU times: user 95 µs, sys: 0 ns, total: 95 µs
Wall time: 96.1 µs
CPU times: user 83 µs, sys: 0 ns, total: 83 µs
Wall time: 84.2 µs
CPU times: user 53 µs, sys: 1e+03 ns, total: 54 µs
Wall time: 53.9 µs
```

11 Disegnare un rettangolo vuoto

[107]:
```
def draw_empty_rectangle(img,x1,y1,x2,y2,colore):
    # dobbiamo disegnare le 4 linee
    draw_h_line(img, x1, y1, x2, colore)
    draw_h_line(img, x1, y2, x2, colore)
    draw_v_line(img, x1, y1, y2, colore)
    draw_v_line(img, x2, y1, y2, colore)

def draw_empty_rectangle2(img,x1,y1,x2,y2,colore):
    for x in range(x1, x2+1):
        draw_pixel(img, x, y1, colore)
        draw_pixel(img, x, y2, colore)
    # oppure disegnare i pixel orizzontali
```

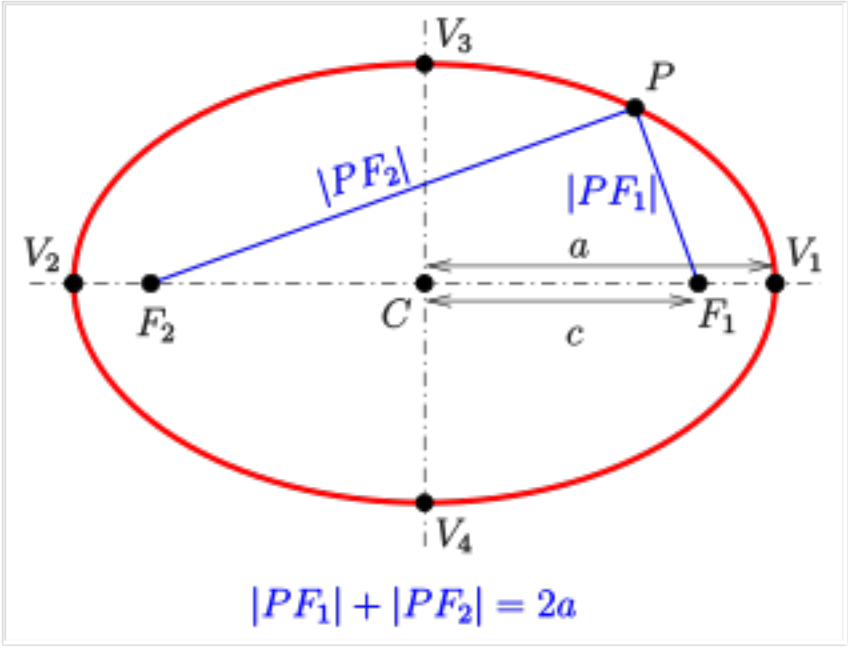
```
    # e poi i verticali
    for y in range(y1, y2+1):
        draw_pixel(img, x1, y, colore)
        draw_pixel(img, x2, y, colore)

img = images.load('3cime.png')
draw_empty_rectangle(img, 50, 50, 100, 150, yellow)

images.visd(img)
```

12 Disegnare un rettangolo pieno

[23]:
```
# questa è facile
def draw_rectangle(img, x1,y1, x2,y2, colore):
    for x in range(x1, x2+1):
        for y in range(y1, y2+1):
            draw_pixel(img, x,y, colore)

img = images.load('3cime.png')
draw_rectangle(img, 30,50, 80, 120, purple)

images.visd(img)
```

13 Disegnare una ellisse

- una ellisse ha la proprietà che: **la somma delle distanze di ogni suo punto dai due fuochi è costante**



[117]:
```
# disegnamo una ellisse PIENA colorando tutti i pixel
# --- con somma delle distanze dai fuochi <= D
from math import sqrt, dist

def draw_ellisse(img, x1, y1, x2, y2, D, colore):
```

```
    larghezza = len(img[0])
    altezza   = len(img)
    # scandisco tutti i pixel della immagine
    for x in range(larghezza):
        for y in range(altezza):
            # per ciascuno calcolo le due distanze dai fuochi
            D1 = dist((x,y), (x1,y1))
            D2 = dist((x,y), (x2,y2))
            # se la somma < D allora il pixel è DENTRO e lo coloro
            if D1+D2 < D:
                img[y][x] = colore
            # altrimenti lo ignoro

            # NOTA: non devo controllare se sono dentro l'immagine
            # perchè scandisco SOLO i pixel della immagine

# NOTA: D deve essere più grande della distanza tra i due fuochi
# TODO: si possono scandire meno pixel invece che tutta l'immagine

img = images.load('3cime.png')
draw_ellisse(img, 50, 100, 90, 200, 110,  cyan)

images.visd(img)
```

[122]:
```
def draw_ellisse_vuota(img, x1, y1, x2, y2, D, colore):
    larghezza = len(img[0])
    altezza   = len(img)
    # scandisco tutti i pixel della immagine
    for x in range(larghezza):
        for y in range(altezza):
```

```
            # per ciascuno calcolo le due distanze dai fuochi
            D1 = dist((x,y), (x1,y1))
            D2 = dist((x,y), (x2,y2))
            # se la somma - D è piccola sono sul bordo e lo coloro
            if abs((D1+D2) - D) < 0.5 :
                img[y][x] = colore
            # altrimenti lo ignoro

# D deve essere più grande della distanza tra i fuochi
img = images.load('3cime.png')
draw_ellisse_vuota(img, 100, 100, 250, 150, 180,  green)

images.visd(img)
```

14 e per disegnare un cerchio?

14.1 "un cerchio è una ellisse che non ce l'ha fatta" :-)

[123]:
```
# cerchio = ellisse con entrambi i fuochi nello stesso punto e somma delle␣
 ↪distanze = 2 raggio
def draw_circle(img, x, y, r, colore):
    draw_ellisse(img, x, y, x, y, 2*r, colore)

def draw_circle_vuoto(img, x, y, r, colore):
    draw_ellisse_vuota(img, x, y, x, y, 2*r, colore)

img = images.load('3cime.png')
draw_circle_vuoto(img, 150, 100, 50,  yellow)
draw_circle(img, 250, 100, 30,  yellow)
```

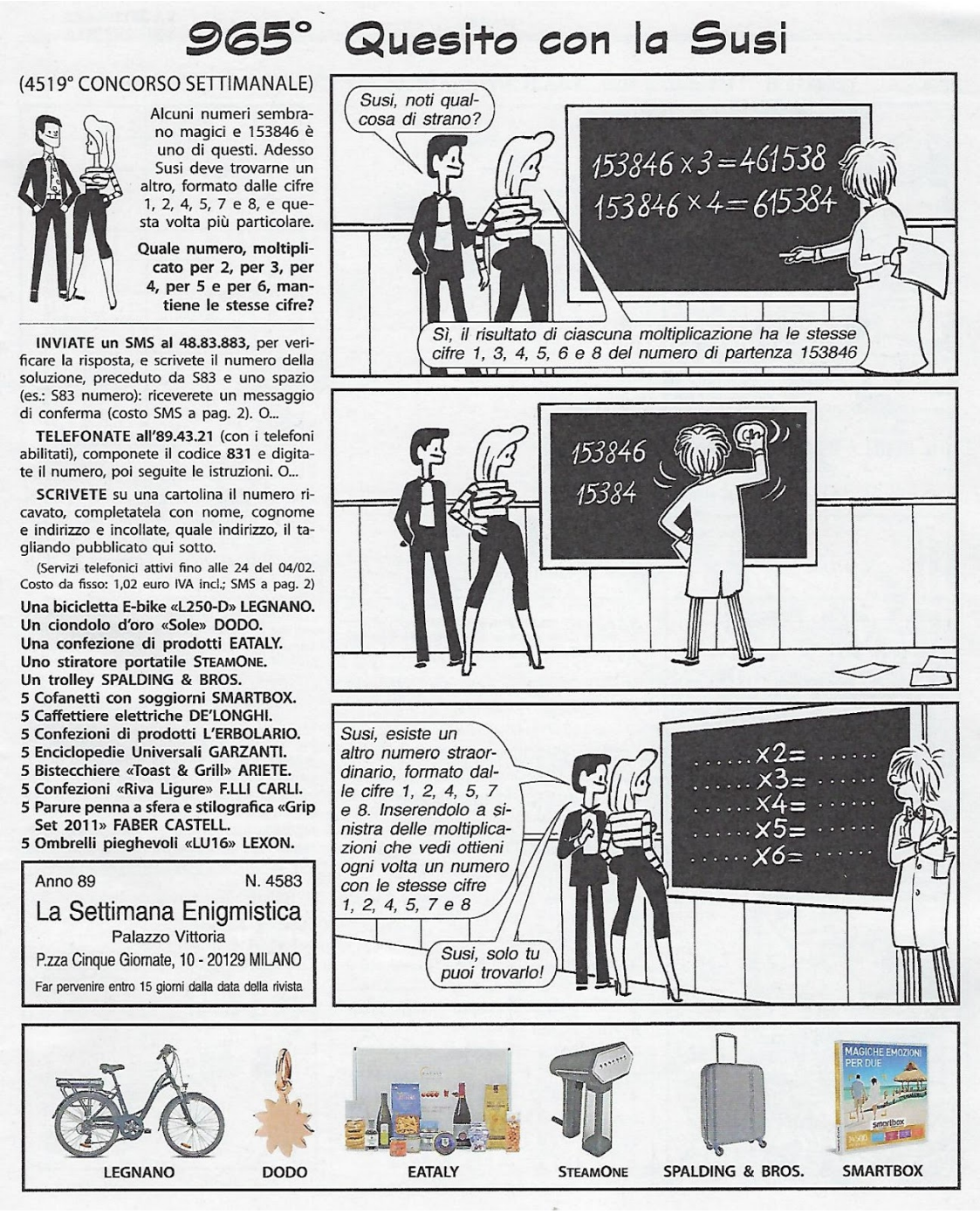
```
images.visd(img)
```

14.2 Come disegnare una parabola? una iperbole? una funzione generica?

per esercizio, basta che vi ricordiate le funzioni e usiate draw_pixel

[27]:
```
# NEXT lesson: manipolazione immagini ed effetti grafici
# gray
# blur
# contrast
# pixellation
# random noise
# lens
# edge
```

15 Quesito con la Susi 965



15.1 Idea di soluzione "brute force" (esplorando tutte le possibili soluzioni)

- trovo tutte le permutazioni delle 6 cifre (sono solo 6! = 720)
- per ciascuna permutazione calcolo i 5 prodotti richiesti
- e controllo che diano tutti come risultato un numero con quelle stesse cifre

Mi serve di: - calcolare un numero data la sequenza di cifre - calcolare il prodotto - spezzare il

risultato nelle sue cifre - controllare che due gruppi di cifre siano uguali

[87]:
```
# per ottenere le permutazioni sfrutto il modulo itertools
from itertools import permutations     # vedremo più in là come generarle noi
cifre = [1, 2, 4, 5, 7, 8]
permutazioni = list(permutations(cifre))
print('Le permutazioni sono:', len(permutazioni),permutazioni,sep='\n')
```

```
Le permutazioni sono:
720
[(1, 2, 4, 5, 7, 8), (1, 2, 4, 5, 8, 7), (1, 2, 4, 7, 5, 8), (1, 2, 4, 7, 8, 5),
(1, 2, 4, 8, 5, 7), (1, 2, 4, 8, 7, 5), (1, 2, 5, 4, 7, 8), (1, 2, 5, 4, 8, 7),
(1, 2, 5, 7, 4, 8), (1, 2, 5, 7, 8, 4), (1, 2, 5, 8, 4, 7), (1, 2, 5, 8, 7, 4),
(1, 2, 7, 4, 5, 8), (1, 2, 7, 4, 8, 5), (1, 2, 7, 5, 4, 8), (1, 2, 7, 5, 8, 4),
(1, 2, 7, 8, 4, 5), (1, 2, 7, 8, 5, 4), (1, 2, 8, 4, 5, 7), (1, 2, 8, 4, 7, 5),
(1, 2, 8, 5, 4, 7), (1, 2, 8, 5, 7, 4), (1, 2, 8, 7, 4, 5), (1, 2, 8, 7, 5, 4),
(1, 4, 2, 5, 7, 8), (1, 4, 2, 5, 8, 7), (1, 4, 2, 7, 5, 8), (1, 4, 2, 7, 8, 5),
(1, 4, 2, 8, 5, 7), (1, 4, 2, 8, 7, 5), (1, 4, 5, 2, 7, 8), (1, 4, 5, 2, 8, 7),
(1, 4, 5, 7, 2, 8), (1, 4, 5, 7, 8, 2), (1, 4, 5, 8, 2, 7), (1, 4, 5, 8, 7, 2),
(1, 4, 7, 2, 5, 8), (1, 4, 7, 2, 8, 5), (1, 4, 7, 5, 2, 8), (1, 4, 7, 5, 8, 2),
(1, 4, 7, 8, 2, 5), (1, 4, 7, 8, 5, 2), (1, 4, 8, 2, 5, 7), (1, 4, 8, 2, 7, 5),
(1, 4, 8, 5, 2, 7), (1, 4, 8, 5, 7, 2), (1, 4, 8, 7, 2, 5), (1, 4, 8, 7, 5, 2),
(1, 5, 2, 4, 7, 8), (1, 5, 2, 4, 8, 7), (1, 5, 2, 7, 4, 8), (1, 5, 2, 7, 8, 4),
(1, 5, 2, 8, 4, 7), (1, 5, 2, 8, 7, 4), (1, 5, 4, 2, 7, 8), (1, 5, 4, 2, 8, 7),
(1, 5, 4, 7, 2, 8), (1, 5, 4, 7, 8, 2), (1, 5, 4, 8, 2, 7), (1, 5, 4, 8, 7, 2),
(1, 5, 7, 2, 4, 8), (1, 5, 7, 2, 8, 4), (1, 5, 7, 4, 2, 8), (1, 5, 7, 4, 8, 2),
(1, 5, 7, 8, 2, 4), (1, 5, 7, 8, 4, 2), (1, 5, 8, 2, 4, 7), (1, 5, 8, 2, 7, 4),
(1, 5, 8, 4, 2, 7), (1, 5, 8, 4, 7, 2), (1, 5, 8, 7, 2, 4), (1, 5, 8, 7, 4, 2),
(1, 7, 2, 4, 5, 8), (1, 7, 2, 4, 8, 5), (1, 7, 2, 5, 4, 8), (1, 7, 2, 5, 8, 4),
(1, 7, 2, 8, 4, 5), (1, 7, 2, 8, 5, 4), (1, 7, 4, 2, 5, 8), (1, 7, 4, 2, 8, 5),
(1, 7, 4, 5, 2, 8), (1, 7, 4, 5, 8, 2), (1, 7, 4, 8, 2, 5), (1, 7, 4, 8, 5, 2),
(1, 7, 5, 2, 4, 8), (1, 7, 5, 2, 8, 4), (1, 7, 5, 4, 2, 8), (1, 7, 5, 4, 8, 2),
(1, 7, 5, 8, 2, 4), (1, 7, 5, 8, 4, 2), (1, 7, 8, 2, 4, 5), (1, 7, 8, 2, 5, 4),
(1, 7, 8, 4, 2, 5), (1, 7, 8, 4, 5, 2), (1, 7, 8, 5, 2, 4), (1, 7, 8, 5, 4, 2),
(1, 8, 2, 4, 5, 7), (1, 8, 2, 4, 7, 5), (1, 8, 2, 5, 4, 7), (1, 8, 2, 5, 7, 4),
(1, 8, 2, 7, 4, 5), (1, 8, 2, 7, 5, 4), (1, 8, 4, 2, 5, 7), (1, 8, 4, 2, 7, 5),
(1, 8, 4, 5, 2, 7), (1, 8, 4, 5, 7, 2), (1, 8, 4, 7, 2, 5), (1, 8, 4, 7, 5, 2),
(1, 8, 5, 2, 4, 7), (1, 8, 5, 2, 7, 4), (1, 8, 5, 4, 2, 7), (1, 8, 5, 4, 7, 2),
(1, 8, 5, 7, 2, 4), (1, 8, 5, 7, 4, 2), (1, 8, 7, 2, 4, 5), (1, 8, 7, 2, 5, 4),
(1, 8, 7, 4, 2, 5), (1, 8, 7, 4, 5, 2), (1, 8, 7, 5, 2, 4), (1, 8, 7, 5, 4, 2),
(2, 1, 4, 5, 7, 8), (2, 1, 4, 5, 8, 7), (2, 1, 4, 7, 5, 8), (2, 1, 4, 7, 8, 5),
(2, 1, 4, 8, 5, 7), (2, 1, 4, 8, 7, 5), (2, 1, 5, 4, 7, 8), (2, 1, 5, 4, 8, 7),
(2, 1, 5, 7, 4, 8), (2, 1, 5, 7, 8, 4), (2, 1, 5, 8, 4, 7), (2, 1, 5, 8, 7, 4),
(2, 1, 7, 4, 5, 8), (2, 1, 7, 4, 8, 5), (2, 1, 7, 5, 4, 8), (2, 1, 7, 5, 8, 4),
(2, 1, 7, 8, 4, 5), (2, 1, 7, 8, 5, 4), (2, 1, 8, 4, 5, 7), (2, 1, 8, 4, 7, 5),
(2, 1, 8, 5, 4, 7), (2, 1, 8, 5, 7, 4), (2, 1, 8, 7, 4, 5), (2, 1, 8, 7, 5, 4),
(2, 4, 1, 5, 7, 8), (2, 4, 1, 5, 8, 7), (2, 4, 1, 7, 5, 8), (2, 4, 1, 7, 8, 5),
(2, 4, 1, 8, 5, 7), (2, 4, 1, 8, 7, 5), (2, 4, 5, 1, 7, 8), (2, 4, 5, 1, 8, 7),
```

(2, 4, 5, 7, 1, 8), (2, 4, 5, 7, 8, 1), (2, 4, 5, 8, 1, 7), (2, 4, 5, 8, 7, 1),
(2, 4, 7, 1, 5, 8), (2, 4, 7, 1, 8, 5), (2, 4, 7, 5, 1, 8), (2, 4, 7, 5, 8, 1),
(2, 4, 7, 8, 1, 5), (2, 4, 7, 8, 5, 1), (2, 4, 8, 1, 5, 7), (2, 4, 8, 1, 7, 5),
(2, 4, 8, 5, 1, 7), (2, 4, 8, 5, 7, 1), (2, 4, 8, 7, 1, 5), (2, 4, 8, 7, 5, 1),
(2, 5, 1, 4, 7, 8), (2, 5, 1, 4, 8, 7), (2, 5, 1, 7, 4, 8), (2, 5, 1, 7, 8, 4),
(2, 5, 1, 8, 4, 7), (2, 5, 1, 8, 7, 4), (2, 5, 4, 1, 7, 8), (2, 5, 4, 1, 8, 7),
(2, 5, 4, 7, 1, 8), (2, 5, 4, 7, 8, 1), (2, 5, 4, 8, 1, 7), (2, 5, 4, 8, 7, 1),
(2, 5, 7, 1, 4, 8), (2, 5, 7, 1, 8, 4), (2, 5, 7, 4, 1, 8), (2, 5, 7, 4, 8, 1),
(2, 5, 7, 8, 1, 4), (2, 5, 7, 8, 4, 1), (2, 5, 8, 1, 4, 7), (2, 5, 8, 1, 7, 4),
(2, 5, 8, 4, 1, 7), (2, 5, 8, 4, 7, 1), (2, 5, 8, 7, 1, 4), (2, 5, 8, 7, 4, 1),
(2, 7, 1, 4, 5, 8), (2, 7, 1, 4, 8, 5), (2, 7, 1, 5, 4, 8), (2, 7, 1, 5, 8, 4),
(2, 7, 1, 8, 4, 5), (2, 7, 1, 8, 5, 4), (2, 7, 4, 1, 5, 8), (2, 7, 4, 1, 8, 5),
(2, 7, 4, 5, 1, 8), (2, 7, 4, 5, 8, 1), (2, 7, 4, 8, 1, 5), (2, 7, 4, 8, 5, 1),
(2, 7, 5, 1, 4, 8), (2, 7, 5, 1, 8, 4), (2, 7, 5, 4, 1, 8), (2, 7, 5, 4, 8, 1),
(2, 7, 5, 8, 1, 4), (2, 7, 5, 8, 4, 1), (2, 7, 8, 1, 4, 5), (2, 7, 8, 1, 5, 4),
(2, 7, 8, 4, 1, 5), (2, 7, 8, 4, 5, 1), (2, 7, 8, 5, 1, 4), (2, 7, 8, 5, 4, 1),
(2, 8, 1, 4, 5, 7), (2, 8, 1, 4, 7, 5), (2, 8, 1, 5, 4, 7), (2, 8, 1, 5, 7, 4),
(2, 8, 1, 7, 4, 5), (2, 8, 1, 7, 5, 4), (2, 8, 4, 1, 5, 7), (2, 8, 4, 1, 7, 5),
(2, 8, 4, 5, 1, 7), (2, 8, 4, 5, 7, 1), (2, 8, 4, 7, 1, 5), (2, 8, 4, 7, 5, 1),
(2, 8, 5, 1, 4, 7), (2, 8, 5, 1, 7, 4), (2, 8, 5, 4, 1, 7), (2, 8, 5, 4, 7, 1),
(2, 8, 5, 7, 1, 4), (2, 8, 5, 7, 4, 1), (2, 8, 7, 1, 4, 5), (2, 8, 7, 1, 5, 4),
(2, 8, 7, 4, 1, 5), (2, 8, 7, 4, 5, 1), (2, 8, 7, 5, 1, 4), (2, 8, 7, 5, 4, 1),
(4, 1, 2, 5, 7, 8), (4, 1, 2, 5, 8, 7), (4, 1, 2, 7, 5, 8), (4, 1, 2, 7, 8, 5),
(4, 1, 2, 8, 5, 7), (4, 1, 2, 8, 7, 5), (4, 1, 5, 2, 7, 8), (4, 1, 5, 2, 8, 7),
(4, 1, 5, 7, 2, 8), (4, 1, 5, 7, 8, 2), (4, 1, 5, 8, 2, 7), (4, 1, 5, 8, 7, 2),
(4, 1, 7, 2, 5, 8), (4, 1, 7, 2, 8, 5), (4, 1, 7, 5, 2, 8), (4, 1, 7, 5, 8, 2),
(4, 1, 7, 8, 2, 5), (4, 1, 7, 8, 5, 2), (4, 1, 8, 2, 5, 7), (4, 1, 8, 2, 7, 5),
(4, 1, 8, 5, 2, 7), (4, 1, 8, 5, 7, 2), (4, 1, 8, 7, 2, 5), (4, 1, 8, 7, 5, 2),
(4, 2, 1, 5, 7, 8), (4, 2, 1, 5, 8, 7), (4, 2, 1, 7, 5, 8), (4, 2, 1, 7, 8, 5),
(4, 2, 1, 8, 5, 7), (4, 2, 1, 8, 7, 5), (4, 2, 5, 1, 7, 8), (4, 2, 5, 1, 8, 7),
(4, 2, 5, 7, 1, 8), (4, 2, 5, 7, 8, 1), (4, 2, 5, 8, 1, 7), (4, 2, 5, 8, 7, 1),
(4, 2, 7, 1, 5, 8), (4, 2, 7, 1, 8, 5), (4, 2, 7, 5, 1, 8), (4, 2, 7, 5, 8, 1),
(4, 2, 7, 8, 1, 5), (4, 2, 7, 8, 5, 1), (4, 2, 8, 1, 5, 7), (4, 2, 8, 1, 7, 5),
(4, 2, 8, 5, 1, 7), (4, 2, 8, 5, 7, 1), (4, 2, 8, 7, 1, 5), (4, 2, 8, 7, 5, 1),
(4, 5, 1, 2, 7, 8), (4, 5, 1, 2, 8, 7), (4, 5, 1, 7, 2, 8), (4, 5, 1, 7, 8, 2),
(4, 5, 1, 8, 2, 7), (4, 5, 1, 8, 7, 2), (4, 5, 2, 1, 7, 8), (4, 5, 2, 1, 8, 7),
(4, 5, 2, 7, 1, 8), (4, 5, 2, 7, 8, 1), (4, 5, 2, 8, 1, 7), (4, 5, 2, 8, 7, 1),
(4, 5, 7, 1, 2, 8), (4, 5, 7, 1, 8, 2), (4, 5, 7, 2, 1, 8), (4, 5, 7, 2, 8, 1),
(4, 5, 7, 8, 1, 2), (4, 5, 7, 8, 2, 1), (4, 5, 8, 1, 2, 7), (4, 5, 8, 1, 7, 2),
(4, 5, 8, 2, 1, 7), (4, 5, 8, 2, 7, 1), (4, 5, 8, 7, 1, 2), (4, 5, 8, 7, 2, 1),
(4, 7, 1, 2, 5, 8), (4, 7, 1, 2, 8, 5), (4, 7, 1, 5, 2, 8), (4, 7, 1, 5, 8, 2),
(4, 7, 1, 8, 2, 5), (4, 7, 1, 8, 5, 2), (4, 7, 2, 1, 5, 8), (4, 7, 2, 1, 8, 5),
(4, 7, 2, 5, 1, 8), (4, 7, 2, 5, 8, 1), (4, 7, 2, 8, 1, 5), (4, 7, 2, 8, 5, 1),
(4, 7, 5, 1, 2, 8), (4, 7, 5, 1, 8, 2), (4, 7, 5, 2, 1, 8), (4, 7, 5, 2, 8, 1),
(4, 7, 5, 8, 1, 2), (4, 7, 5, 8, 2, 1), (4, 7, 8, 1, 2, 5), (4, 7, 8, 1, 5, 2),
(4, 7, 8, 2, 1, 5), (4, 7, 8, 2, 5, 1), (4, 7, 8, 5, 1, 2), (4, 7, 8, 5, 2, 1),
(4, 8, 1, 2, 5, 7), (4, 8, 1, 2, 7, 5), (4, 8, 1, 5, 2, 7), (4, 8, 1, 5, 7, 2),
(4, 8, 1, 7, 2, 5), (4, 8, 1, 7, 5, 2), (4, 8, 2, 1, 5, 7), (4, 8, 2, 1, 7, 5),

(4, 8, 2, 5, 1, 7), (4, 8, 2, 5, 7, 1), (4, 8, 2, 7, 1, 5), (4, 8, 2, 7, 5, 1),
(4, 8, 5, 1, 2, 7), (4, 8, 5, 1, 7, 2), (4, 8, 5, 2, 1, 7), (4, 8, 5, 2, 7, 1),
(4, 8, 5, 7, 1, 2), (4, 8, 5, 7, 2, 1), (4, 8, 7, 1, 2, 5), (4, 8, 7, 1, 5, 2),
(4, 8, 7, 2, 1, 5), (4, 8, 7, 2, 5, 1), (4, 8, 7, 5, 1, 2), (4, 8, 7, 5, 2, 1),
(5, 1, 2, 4, 7, 8), (5, 1, 2, 4, 8, 7), (5, 1, 2, 7, 4, 8), (5, 1, 2, 7, 8, 4),
(5, 1, 2, 8, 4, 7), (5, 1, 2, 8, 7, 4), (5, 1, 4, 2, 7, 8), (5, 1, 4, 2, 8, 7),
(5, 1, 4, 7, 2, 8), (5, 1, 4, 7, 8, 2), (5, 1, 4, 8, 2, 7), (5, 1, 4, 8, 7, 2),
(5, 1, 7, 2, 4, 8), (5, 1, 7, 2, 8, 4), (5, 1, 7, 4, 2, 8), (5, 1, 7, 4, 8, 2),
(5, 1, 7, 8, 2, 4), (5, 1, 7, 8, 4, 2), (5, 1, 8, 2, 4, 7), (5, 1, 8, 2, 7, 4),
(5, 1, 8, 4, 2, 7), (5, 1, 8, 4, 7, 2), (5, 1, 8, 7, 2, 4), (5, 1, 8, 7, 4, 2),
(5, 2, 1, 4, 7, 8), (5, 2, 1, 4, 8, 7), (5, 2, 1, 7, 4, 8), (5, 2, 1, 7, 8, 4),
(5, 2, 1, 8, 4, 7), (5, 2, 1, 8, 7, 4), (5, 2, 4, 1, 7, 8), (5, 2, 4, 1, 8, 7),
(5, 2, 4, 7, 1, 8), (5, 2, 4, 7, 8, 1), (5, 2, 4, 8, 1, 7), (5, 2, 4, 8, 7, 1),
(5, 2, 7, 1, 4, 8), (5, 2, 7, 1, 8, 4), (5, 2, 7, 4, 1, 8), (5, 2, 7, 4, 8, 1),
(5, 2, 7, 8, 1, 4), (5, 2, 7, 8, 4, 1), (5, 2, 8, 1, 4, 7), (5, 2, 8, 1, 7, 4),
(5, 2, 8, 4, 1, 7), (5, 2, 8, 4, 7, 1), (5, 2, 8, 7, 1, 4), (5, 2, 8, 7, 4, 1),
(5, 4, 1, 2, 7, 8), (5, 4, 1, 2, 8, 7), (5, 4, 1, 7, 2, 8), (5, 4, 1, 7, 8, 2),
(5, 4, 1, 8, 2, 7), (5, 4, 1, 8, 7, 2), (5, 4, 2, 1, 7, 8), (5, 4, 2, 1, 8, 7),
(5, 4, 2, 7, 1, 8), (5, 4, 2, 7, 8, 1), (5, 4, 2, 8, 1, 7), (5, 4, 2, 8, 7, 1),
(5, 4, 7, 1, 2, 8), (5, 4, 7, 1, 8, 2), (5, 4, 7, 2, 1, 8), (5, 4, 7, 2, 8, 1),
(5, 4, 7, 8, 1, 2), (5, 4, 7, 8, 2, 1), (5, 4, 8, 1, 2, 7), (5, 4, 8, 1, 7, 2),
(5, 4, 8, 2, 1, 7), (5, 4, 8, 2, 7, 1), (5, 4, 8, 7, 1, 2), (5, 4, 8, 7, 2, 1),
(5, 7, 1, 2, 4, 8), (5, 7, 1, 2, 8, 4), (5, 7, 1, 4, 2, 8), (5, 7, 1, 4, 8, 2),
(5, 7, 1, 8, 2, 4), (5, 7, 1, 8, 4, 2), (5, 7, 2, 1, 4, 8), (5, 7, 2, 1, 8, 4),
(5, 7, 2, 4, 1, 8), (5, 7, 2, 4, 8, 1), (5, 7, 2, 8, 1, 4), (5, 7, 2, 8, 4, 1),
(5, 7, 4, 1, 2, 8), (5, 7, 4, 1, 8, 2), (5, 7, 4, 2, 1, 8), (5, 7, 4, 2, 8, 1),
(5, 7, 4, 8, 1, 2), (5, 7, 4, 8, 2, 1), (5, 7, 8, 1, 2, 4), (5, 7, 8, 1, 4, 2),
(5, 7, 8, 2, 1, 4), (5, 7, 8, 2, 4, 1), (5, 7, 8, 4, 1, 2), (5, 7, 8, 4, 2, 1),
(5, 8, 1, 2, 4, 7), (5, 8, 1, 2, 7, 4), (5, 8, 1, 4, 2, 7), (5, 8, 1, 4, 7, 2),
(5, 8, 1, 7, 2, 4), (5, 8, 1, 7, 4, 2), (5, 8, 2, 1, 4, 7), (5, 8, 2, 1, 7, 4),
(5, 8, 2, 4, 1, 7), (5, 8, 2, 4, 7, 1), (5, 8, 2, 7, 1, 4), (5, 8, 2, 7, 4, 1),
(5, 8, 4, 1, 2, 7), (5, 8, 4, 1, 7, 2), (5, 8, 4, 2, 1, 7), (5, 8, 4, 2, 7, 1),
(5, 8, 4, 7, 1, 2), (5, 8, 4, 7, 2, 1), (5, 8, 7, 1, 2, 4), (5, 8, 7, 1, 4, 2),
(5, 8, 7, 2, 1, 4), (5, 8, 7, 2, 4, 1), (5, 8, 7, 4, 1, 2), (5, 8, 7, 4, 2, 1),
(7, 1, 2, 4, 5, 8), (7, 1, 2, 4, 8, 5), (7, 1, 2, 5, 4, 8), (7, 1, 2, 5, 8, 4),
(7, 1, 2, 8, 4, 5), (7, 1, 2, 8, 5, 4), (7, 1, 4, 2, 5, 8), (7, 1, 4, 2, 8, 5),
(7, 1, 4, 5, 2, 8), (7, 1, 4, 5, 8, 2), (7, 1, 4, 8, 2, 5), (7, 1, 4, 8, 5, 2),
(7, 1, 5, 2, 4, 8), (7, 1, 5, 2, 8, 4), (7, 1, 5, 4, 2, 8), (7, 1, 5, 4, 8, 2),
(7, 1, 5, 8, 2, 4), (7, 1, 5, 8, 4, 2), (7, 1, 8, 2, 4, 5), (7, 1, 8, 2, 5, 4),
(7, 1, 8, 4, 2, 5), (7, 1, 8, 4, 5, 2), (7, 1, 8, 5, 2, 4), (7, 1, 8, 5, 4, 2),
(7, 2, 1, 4, 5, 8), (7, 2, 1, 4, 8, 5), (7, 2, 1, 5, 4, 8), (7, 2, 1, 5, 8, 4),
(7, 2, 1, 8, 4, 5), (7, 2, 1, 8, 5, 4), (7, 2, 4, 1, 5, 8), (7, 2, 4, 1, 8, 5),
(7, 2, 4, 5, 1, 8), (7, 2, 4, 5, 8, 1), (7, 2, 4, 8, 1, 5), (7, 2, 4, 8, 5, 1),
(7, 2, 5, 1, 4, 8), (7, 2, 5, 1, 8, 4), (7, 2, 5, 4, 1, 8), (7, 2, 5, 4, 8, 1),
(7, 2, 5, 8, 1, 4), (7, 2, 5, 8, 4, 1), (7, 2, 8, 1, 4, 5), (7, 2, 8, 1, 5, 4),
(7, 2, 8, 4, 1, 5), (7, 2, 8, 4, 5, 1), (7, 2, 8, 5, 1, 4), (7, 2, 8, 5, 4, 1),
(7, 4, 1, 2, 5, 8), (7, 4, 1, 2, 8, 5), (7, 4, 1, 5, 2, 8), (7, 4, 1, 5, 8, 2),
(7, 4, 1, 8, 2, 5), (7, 4, 1, 8, 5, 2), (7, 4, 2, 1, 5, 8), (7, 4, 2, 1, 8, 5),

(7, 4, 2, 5, 1, 8), (7, 4, 2, 5, 8, 1), (7, 4, 2, 8, 1, 5), (7, 4, 2, 8, 5, 1),
(7, 4, 5, 1, 2, 8), (7, 4, 5, 1, 8, 2), (7, 4, 5, 2, 1, 8), (7, 4, 5, 2, 8, 1),
(7, 4, 5, 8, 1, 2), (7, 4, 5, 8, 2, 1), (7, 4, 8, 1, 2, 5), (7, 4, 8, 1, 5, 2),
(7, 4, 8, 2, 1, 5), (7, 4, 8, 2, 5, 1), (7, 4, 8, 5, 1, 2), (7, 4, 8, 5, 2, 1),
(7, 5, 1, 2, 4, 8), (7, 5, 1, 2, 8, 4), (7, 5, 1, 4, 2, 8), (7, 5, 1, 4, 8, 2),
(7, 5, 1, 8, 2, 4), (7, 5, 1, 8, 4, 2), (7, 5, 2, 1, 4, 8), (7, 5, 2, 1, 8, 4),
(7, 5, 2, 4, 1, 8), (7, 5, 2, 4, 8, 1), (7, 5, 2, 8, 1, 4), (7, 5, 2, 8, 4, 1),
(7, 5, 4, 1, 2, 8), (7, 5, 4, 1, 8, 2), (7, 5, 4, 2, 1, 8), (7, 5, 4, 2, 8, 1),
(7, 5, 4, 8, 1, 2), (7, 5, 4, 8, 2, 1), (7, 5, 8, 1, 2, 4), (7, 5, 8, 1, 4, 2),
(7, 5, 8, 2, 1, 4), (7, 5, 8, 2, 4, 1), (7, 5, 8, 4, 1, 2), (7, 5, 8, 4, 2, 1),
(7, 8, 1, 2, 4, 5), (7, 8, 1, 2, 5, 4), (7, 8, 1, 4, 2, 5), (7, 8, 1, 4, 5, 2),
(7, 8, 1, 5, 2, 4), (7, 8, 1, 5, 4, 2), (7, 8, 2, 1, 4, 5), (7, 8, 2, 1, 5, 4),
(7, 8, 2, 4, 1, 5), (7, 8, 2, 4, 5, 1), (7, 8, 2, 5, 1, 4), (7, 8, 2, 5, 4, 1),
(7, 8, 4, 1, 2, 5), (7, 8, 4, 1, 5, 2), (7, 8, 4, 2, 1, 5), (7, 8, 4, 2, 5, 1),
(7, 8, 4, 5, 1, 2), (7, 8, 4, 5, 2, 1), (7, 8, 5, 1, 2, 4), (7, 8, 5, 1, 4, 2),
(7, 8, 5, 2, 1, 4), (7, 8, 5, 2, 4, 1), (7, 8, 5, 4, 1, 2), (7, 8, 5, 4, 2, 1),
(8, 1, 2, 4, 5, 7), (8, 1, 2, 4, 7, 5), (8, 1, 2, 5, 4, 7), (8, 1, 2, 5, 7, 4),
(8, 1, 2, 7, 4, 5), (8, 1, 2, 7, 5, 4), (8, 1, 4, 2, 5, 7), (8, 1, 4, 2, 7, 5),
(8, 1, 4, 5, 2, 7), (8, 1, 4, 5, 7, 2), (8, 1, 4, 7, 2, 5), (8, 1, 4, 7, 5, 2),
(8, 1, 5, 2, 4, 7), (8, 1, 5, 2, 7, 4), (8, 1, 5, 4, 2, 7), (8, 1, 5, 4, 7, 2),
(8, 1, 5, 7, 2, 4), (8, 1, 5, 7, 4, 2), (8, 1, 7, 2, 4, 5), (8, 1, 7, 2, 5, 4),
(8, 1, 7, 4, 2, 5), (8, 1, 7, 4, 5, 2), (8, 1, 7, 5, 2, 4), (8, 1, 7, 5, 4, 2),
(8, 2, 1, 4, 5, 7), (8, 2, 1, 4, 7, 5), (8, 2, 1, 5, 4, 7), (8, 2, 1, 5, 7, 4),
(8, 2, 1, 7, 4, 5), (8, 2, 1, 7, 5, 4), (8, 2, 4, 1, 5, 7), (8, 2, 4, 1, 7, 5),
(8, 2, 4, 5, 1, 7), (8, 2, 4, 5, 7, 1), (8, 2, 4, 7, 1, 5), (8, 2, 4, 7, 5, 1),
(8, 2, 5, 1, 4, 7), (8, 2, 5, 1, 7, 4), (8, 2, 5, 4, 1, 7), (8, 2, 5, 4, 7, 1),
(8, 2, 5, 7, 1, 4), (8, 2, 5, 7, 4, 1), (8, 2, 7, 1, 4, 5), (8, 2, 7, 1, 5, 4),
(8, 2, 7, 4, 1, 5), (8, 2, 7, 4, 5, 1), (8, 2, 7, 5, 1, 4), (8, 2, 7, 5, 4, 1),
(8, 4, 1, 2, 5, 7), (8, 4, 1, 2, 7, 5), (8, 4, 1, 5, 2, 7), (8, 4, 1, 5, 7, 2),
(8, 4, 1, 7, 2, 5), (8, 4, 1, 7, 5, 2), (8, 4, 2, 1, 5, 7), (8, 4, 2, 1, 7, 5),
(8, 4, 2, 5, 1, 7), (8, 4, 2, 5, 7, 1), (8, 4, 2, 7, 1, 5), (8, 4, 2, 7, 5, 1),
(8, 4, 5, 1, 2, 7), (8, 4, 5, 1, 7, 2), (8, 4, 5, 2, 1, 7), (8, 4, 5, 2, 7, 1),
(8, 4, 5, 7, 1, 2), (8, 4, 5, 7, 2, 1), (8, 4, 7, 1, 2, 5), (8, 4, 7, 1, 5, 2),
(8, 4, 7, 2, 1, 5), (8, 4, 7, 2, 5, 1), (8, 4, 7, 5, 1, 2), (8, 4, 7, 5, 2, 1),
(8, 5, 1, 2, 4, 7), (8, 5, 1, 2, 7, 4), (8, 5, 1, 4, 2, 7), (8, 5, 1, 4, 7, 2),
(8, 5, 1, 7, 2, 4), (8, 5, 1, 7, 4, 2), (8, 5, 2, 1, 4, 7), (8, 5, 2, 1, 7, 4),
(8, 5, 2, 4, 1, 7), (8, 5, 2, 4, 7, 1), (8, 5, 2, 7, 1, 4), (8, 5, 2, 7, 4, 1),
(8, 5, 4, 1, 2, 7), (8, 5, 4, 1, 7, 2), (8, 5, 4, 2, 1, 7), (8, 5, 4, 2, 7, 1),
(8, 5, 4, 7, 1, 2), (8, 5, 4, 7, 2, 1), (8, 5, 7, 1, 2, 4), (8, 5, 7, 1, 4, 2),
(8, 5, 7, 2, 1, 4), (8, 5, 7, 2, 4, 1), (8, 5, 7, 4, 1, 2), (8, 5, 7, 4, 2, 1),
(8, 7, 1, 2, 4, 5), (8, 7, 1, 2, 5, 4), (8, 7, 1, 4, 2, 5), (8, 7, 1, 4, 5, 2),
(8, 7, 1, 5, 2, 4), (8, 7, 1, 5, 4, 2), (8, 7, 2, 1, 4, 5), (8, 7, 2, 1, 5, 4),
(8, 7, 2, 4, 1, 5), (8, 7, 2, 4, 5, 1), (8, 7, 2, 5, 1, 4), (8, 7, 2, 5, 4, 1),
(8, 7, 4, 1, 2, 5), (8, 7, 4, 1, 5, 2), (8, 7, 4, 2, 1, 5), (8, 7, 4, 2, 5, 1),
(8, 7, 4, 5, 1, 2), (8, 7, 4, 5, 2, 1), (8, 7, 5, 1, 2, 4), (8, 7, 5, 1, 4, 2),
(8, 7, 5, 2, 1, 4), (8, 7, 5, 2, 4, 1), (8, 7, 5, 4, 1, 2), (8, 7, 5, 4, 2, 1)]

[124]:
```
Sequenza = list[int] | tuple[int,...]     # voglio usare tuple oppure liste
def calcola(cifre : Sequenza) -> int:
    'per convertire le cifre in numero passo per la stringa corrisponente'
    return int(''.join(map(str,cifre))) # concateno e converto in intero

def scomponi(numero : int) -> Sequenza:
    'per scomporre un numero nelle sue cifre passo per la stringa'
    return list(map(int, str(numero)))  # converto tutto in stringa e poi in␣
 ↪intero un carattere per volta

# Esempio
calcola((4, 7, 2, 5, 1)), scomponi(123456)
```

[124]:
```
(47251, [1, 2, 3, 4, 5, 6])
```

[126]:
```
def controlla(permutazione : Sequenza, X : int, verbose : bool=False) -> bool:
    'controllo che il prodotto della permutazione per X dia le stesse cifre'
    N = calcola(permutazione)                       # ottengo il numero
    cifre = scomponi(N*X)                           # moltiplico e scompongo
    if verbose: print(f"{N} x {X} = {N*X}")         # se voglio visualizzo il␣
 ↪prodotto
    return sorted(cifre) == sorted(permutazione)  # verifico che siano le␣
 ↪stesse cifre (i set non vanno bene)

# esempi presi dal quesito della Susi
controlla([1, 5, 3, 8, 4, 6], 3, True), controlla([1, 5, 3, 8, 4, 6], 4, True)
```

```
153846 x 3 = 461538
153846 x 4 = 615384
```

[126]:
```
(True, True)
```

[130]:
```
%%time
# e finalmente controlliamo tutte le permutazioni
for p in permutazioni:
    prodotti = [2, 3, 4, 5, 6]
    if all( controlla(p,X) for X in prodotti ):
        for X in prodotti:
            controlla(p,X,True)   # stampo il prodotto

# la soluzione è unica!
```

```
142857 x 2 = 285714
142857 x 3 = 428571
142857 x 4 = 571428
142857 x 5 = 714285
142857 x 6 = 857142
CPU times: user 6.24 ms, sys: 168 µs, total: 6.41 ms
```

```
Wall time: 6.54 ms
```

15.2 In realtà potremmo dedurre alcune cose e restringere l'esplorazione

Le cifre disponibili sono [1, 2, 4, 5, 7, 8] - ABCDEF * 5 = numero che finisce per 5 o per 0 - 0 non è presente, il risultato finisce per forza con 5 - F = deve essere dispari (1,5,7) - ABCDEF * 2 = numero pari (2,4,8) - ABCDEF * 4 = numero pari (2,4,8) - ABCDEF * 6 = numero pari (2,4,8) non ammette F=1 o F=5 che darebbero 0 oppure 6 che non sono presenti - quindi **F=7** - ABCDEF * 3 = numero che finisce per 1 OK

Inoltre **A** deve essere **1** altrimenti **A*6** dà riporto

Quindi sappiamo che **F=7 e A=1** per cui possiamo rendere la ricerca più veloce esaminando solo le 24 permutazioni delle altre 4 cifre [4,2,8,5]

[131]:
```
%%time
for p in permutations([2,4,8,5]):
    cifre = [1, *p, 7]
    if all( controlla(cifre,X) for X in [2,3,4,5,6] ):
        for X in prodotti:
            controlla(cifre,X,True)
```

```
142857 x 2 = 285714
142857 x 3 = 428571
142857 x 4 = 571428
142857 x 5 = 714285
142857 x 6 = 857142
CPU times: user 625 µs, sys: 79 µs, total: 704 µs
Wall time: 695 µs
```