Politecnico di Torino

Dipartimento di Automatica e Informatica

Laboratorio 08

LISTE E TABELLE

This Photo by Unknown Author is licensed under CC BY-SA

Esercizio 1

Esercizio 1. Scrivete la funzione `merge(a, b)` che fonde due liste, alternando un elemento della prima e un elemento della seconda. Se una lista è più corta dell'altra, gli elementi vengono alternati fin quando è possibile, poi gli elementi rimasti nella lista più lunga vengono aggiunti ordinatamente in fondo. Se, ad esempio, il contenuto di `a` è `1 4 9 16` e il contenuto di `b` è `9 7 4 9 11`, l'invocazione `merge(a, b)` restituisce una nuova lista contenente i valori `1 9 4 7 9 4 16 9 11`. [P6.30]

Esercizio 1 – *il codice Python*

```
def main():
    # Set up sample lists.
    a = [1, 4, 9, 16]
    b = [9, 7, 4, 9, 11, 0]

    # Demonstrate that merge works correctly.
    print("List a is", a)
    print("List b is", b)
    result = merge(a, b)
    print("The merged list is", result)

    print("With reversed order:")
    result = merge(b, a)
    print("The merged list is", result)
```

```
def merge(a, b):
    """
    Merge two lists, alternating elements from each

    :param a: the first list to take elements from
    :param b: the second list to take elements from
    :return: the merged list
    """
    result = []

    # Merge elements from both lists as long as there are elements in both.
    len_shorter = min(len(a), len(b))
    for i in range(0, len_shorter):
        result.append(a[i])
        result.append(b[i])

    # Add the remaining elements from whichever list was longer.
    # Note: one of the two 'for' loops will NOT be executed, because either a or b has
    # already run out of elements.
    # If they have the same length, then no loop is executed, at all.
    for i in range(len_shorter, len(a)):
        result.append(a[i])
    for i in range(len_shorter, len(b)):
        result.append(b[i])

    return result

# Call the main function.
main()
```

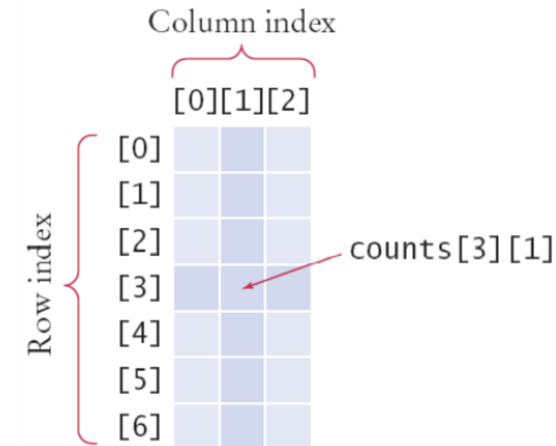
Esercizio 1 – *il codice Python, soluzione alternativa*

```python
def main():
    # Set up sample lists.
    a = [1, 4, 9, 16]
    b = [9, 7, 4, 9, 11, 0]

    # Demonstrate that merge works correctly.
    print("List a is", a)
    print("List b is", b)
    result = merge(a, b)
    print("The merged list is", result)

    print("With reversed order:")
    result = merge(b, a)
    print("The merged list is", result)
```

```python
# Alternative solution with the slice
def merge(a, b):

    l=min(len(a),len(b))
    c=[0]*l*2
    c[::2]=a[:l]
    c[1::2]=b[:l]

    c.extend(a[l:]+b[l:])

    return c

# Call the main function.
main()
```

Dalla teoria... le tabelle

Accedere agli elementi



- Si usano due indici:
  - prima l'indice di riga, poi di colonna

```
medalCount = counts[3][1]
```

- Per stampare
  - Usare il ciclo `for` annidato
  - Ciclo esterno sulle righe (`i`), ciclo interno sulle colonne (`j`)

```
for i in range(COUNTRIES):
    # processa l'i-esima riga
    for j in range(MEDALS) :
        # processa la j-esima Colonna della i-esima riga
        print("%8d" % counts[i][j], end="")
    print() # va a capo alla fine della riga
```

```
       0       3       0
       0       0       1
       0       0       1
       1       0       0
       0       0       1
       3       1       1
       0       1       0
       1       0       1
```

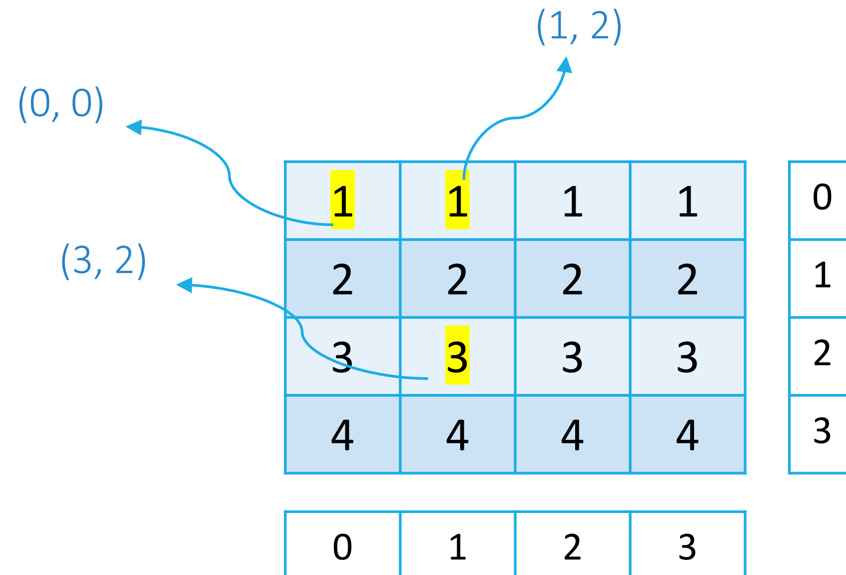
Esercizio 2

Esercizio 2. Scrivete la funzione neighborAverage(values, row, column) che, in una tabella, calcoli il valore medio dei vicini di un elemento nelle otto direzioni, come si può vedere nella figura sotto. Se, però, l'elemento si trova su un bordo della tabella, la media va calcolata considerando soltanto i vicini che appartengono effettivamente alla tabella. Ad esempio, se row e column valgono entrambe 0, ci sono soltanto tre vicini. [P6.23]

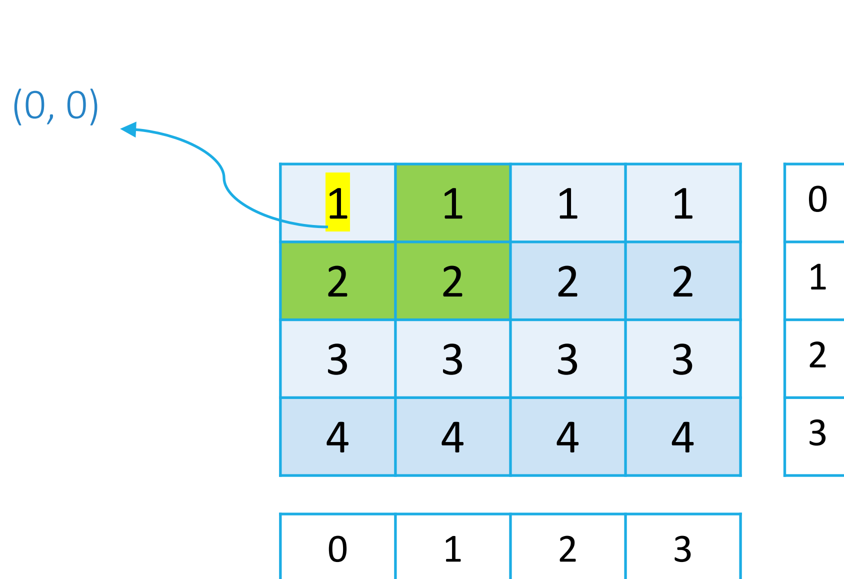
[i -1] [j - 1]	[i - 1] [j]	[i - 1] [j + 1]
[i] [j - 1]	[i] [j]	[i] [j + 1]
[i + 1] [j - 1]	[i + 1] [j]	[i + 1] [j + 1]

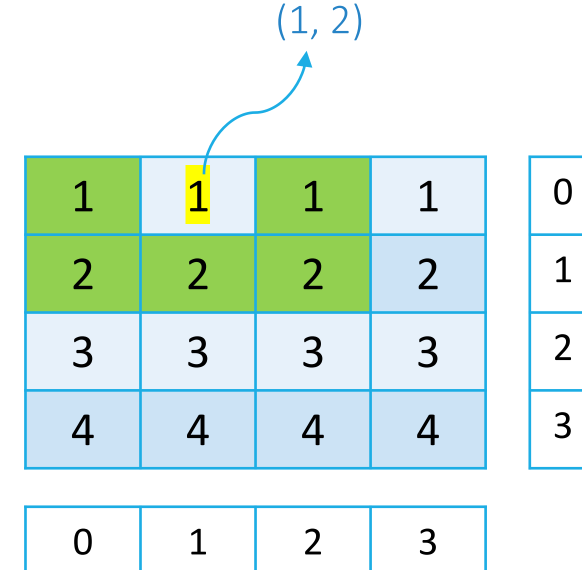
Esercizio 2 – *la tabella d'esempio*

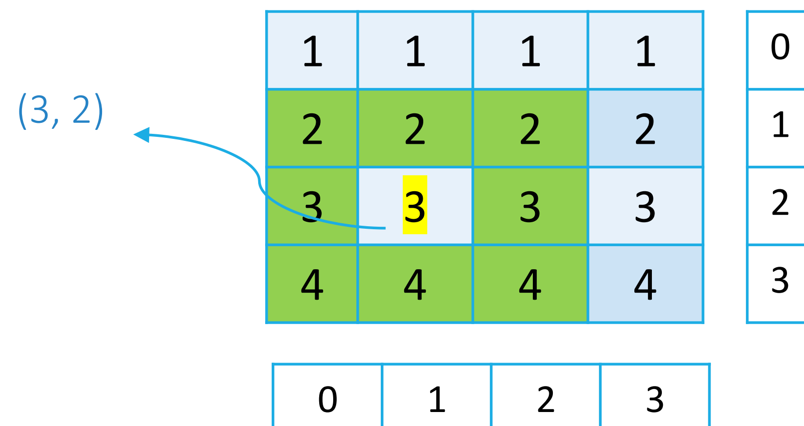
```
# Demonstrate that neighborAverage works correctly.
print("The average at (0,0) is", neighborAverage(table, 0, 0))
print("The average at (1,2) is", neighborAverage(table, 1, 2))
print("The average at (3,2) is", neighborAverage(table, 3, 2))
```



Esercizio 2 – *la tabella d'esempio, capire gli indici*







In *verde* gli *elementi* dei quali calcolare la media nei tre casi proposti.

Esercizio 2 – *il codice Python*

```
def main():
    # Setup a sample table.
    table = [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]]

    # Alternatively, setup a sample table using list comprehensions
    # w, h = 4, 4
    # table = [[y + 1 for x in range(w)] for y in range(h)]

    # print(table)
    for row in table:
        print('|'.join([f'{item:^4d}' for item in row]))  # create a temp list of string-formatted numbers

    # Demonstrate that neighborAverage works correctly.
    print("The average at (0,0) is", neighborAverage(table, 0, 0))
    print("The average at (1,2) is", neighborAverage(table, 1, 2))
    print("The average at (3,2) is", neighborAverage(table, 3, 2))
```

Esercizio 2 – *il codice Python*

```python
def neighborAverage(values, row, column):
    """
    Compute the average of neighbors in a table, handling boundary conditions

    :param values: the table of values to process
    :param row: the row of the element
    :param column: the column of the element
    :return: the average
    """
    count = 0
    total = 0
    # iterate (i,j) over a 3x3 square, centered in (row, column)
    for i in range(row - 1, row + 2):
        for j in range(column - 1, column + 2):
            # I must count the element if:
            # - it is not the "center" element itself
            # - it is not outside the table
            if (i, j) != (row, column) and 0 <= i < len(values) and 0 <= j < len(values[i]):
                total = total + values[i][j]
                count = count + 1

    return total / count


# Call the main function.
main()
```

Esercizio 3

Esercizio 3. *Quadrati magici*. Una matrice $n \times n$ contenente i numeri interi 1, 2, 3, ..., n^2 è un quadrato magico se la somma dei suoi elementi in ciascuna riga, in ciascuna colonna e nelle due diagonali ha lo stesso valore. Ad esempio, questo è un quadrato magico di dimensione 4:

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Scrivete un programma che acquisisca in ingresso 16 valori e verifichi se, dopo averli disposti in una tabella 4 × 4 ordinatamente per righe, da sinistra a destra e dall'alto in basso, formano un quadrato magico. Dovete verificare due proprietà:

1. Nei dati acquisiti sono presenti tutti i numeri 1, 2, ..., 16?
2. Quando i numeri vengono disposti nella tabella, le somme delle righe, delle colonne e delle diagonali sono tutte uguali l'una all'altra?

[P6.21]

Esercizio 3 – *il codice Python*

```python
# Read the values from the user into a 2D board.
def input_board():
    board = [] # table

    for r in range(0, 4): # rows
        row=[]
        for c in range(0, 4): # columns
            val = input(f"Enter a value (position {r + 1},{c + 1}): ")
            row.append(int(val)) # fills the list for the row
        board.append(row)
    return board


def display_board(board):
    """
    Display the board

    :param board: board to be displayed
    :return: None
    """
    for row in range(0, len(board)):
        for col in range(0, len(board[row])):
            print(f'{board[row][col]:3}', end=" ")
        print()
```

Esercizio 3 – *il codice Python*

```python
def check_magic(board):
    # Determine the target total by summing the first row.
    target = sum(board[0])

    # Assume it is a magic square.  Change isMagicSquare to False if we find
    # evidence that it is not.
    isMagicSquare = True

    # Check that all of the numbers are present.
    for i in range(1, 17):
        if i not in board[0] and i not in board[1] and \
                i not in board[2] and i not in board[3]:
            isMagicSquare = False
```

```python
    # Check the rows.
    for i in range(1, len(board)):
        if sum(board[i]) != target:
            isMagicSquare = False

    # Check the columns.
    for j in range(0, 3):
        if board[0][j] + board[1][j] + board[2][j] + board[3][j] != target:
            isMagicSquare = False
```

```python
# Check the diagonals.
if board[0][0] + board[1][1] + board[2][2] + board[3][3] != target:
    isMagicSquare = False
if board[0][3] + board[1][2] + board[2][1] + board[3][0] != target:
    isMagicSquare = False
```

Esercizio 3 – *il codice Python*

```
# Alternative way to check the diagonals using loops
sum_diag = 0
for i in range(len(board)):
    sum_diag = sum_diag + board[i][i]
if sum_diag != target:
    isMagicSquare = False

sum_diag = 0
for i in range(len(board)):
    sum_diag = sum_diag + board[i][len(board) - 1 - i]
if sum_diag != target:
    isMagicSquare = False

return isMagicSquare
```

```
def main():
    board = input_board()
    display_board(board)
    isMagicSquare = check_magic(board)
    # Display the result.
    if isMagicSquare:
        print("It is a magic square.")
    else:
        print("It is not a magic square.")


# run the program
main()
```

Esercizio 4

Esercizio 4. Scrivete la funzione `mergeSorted(a, b)` che fonde due liste **ordinate** (si supponga quindi che siano già ordinate), restituendo una nuova lista **ordinata**. Gestite un indice corrente per ciascuna lista, in modo da tenere traccia della porzione già elaborata. Le liste di partenza non devono essere modificate. Se, ad esempio, il contenuto di `a` è `1 4 9 16` e il contenuto di `b` è `4 7 9 9 11`, l'invocazione `mergeSorted(a, b)` restituisce una nuova lista contenente i valori `1 4 4 7 9 9 9 11 16`. Non utilizzare il metodo sort né la funzione sorted (sfruttare l'informazione che gli elementi di ciascuna lista sono già ordinati per ottenere una soluzione più efficiente). [P6.31]

Esercizio 4 – *il codice Python*

Esercizio 4 – *il codice Python*

```python
def main():
    # Set up two sample lists.
    a = [1, 4, 9, 16]
    b = [4, 7, 9, 9, 11, 19]

    # Demonstrate that mergeSorted works correctly.
    print("List a is", a)
    print("List b is", b)
    merged = mergeSorted(a, b)
    print("The merged list is", merged)

    print("In reverse order, the result should be the same:")
    merged = mergeSorted(b, a)
    print("The merged list is", merged)
```

```python
def mergeSorted(a, b):
    """
    Merge two sorted lists into a single sorted list

    :param a: the first sorted list to merge
    :param b: the second sorted list to merge
    :return: the merged list (sorted)
    """
    a_pos = 0
    b_pos = 0
    result = []

    # As long as there is an unprocessed element in either list.
    while a_pos < len(a) or b_pos < len(b):
        # If there are elements in both lists then take an element from the
        # list that starts with the smaller element.
        if a_pos < len(a) and b_pos < len(b):
            if a[a_pos] <= b[b_pos]:
                result.append(a[a_pos])
                a_pos = a_pos + 1
            else:
                result.append(b[b_pos])
                b_pos = b_pos + 1
        # If only list 'a' has elements then process its first element.
        elif a_pos < len(a):
            result.append(a[a_pos])
            a_pos = a_pos + 1
        # If only list 'b' has elements then process its first element.
        else:
            result.append(b[b_pos])
            b_pos = b_pos + 1

    return result


# Call the main function.
main()
```

Esercizio 5

Esercizio 5. Scrivete un programma che giochi a tic-tac-toe (in italiano, *tris* o *schiera*). Il tic-tac-toe si gioca su una scacchiera 3 × 3, come quella qui raffigurata, da due giocatori che, a turno, posizionano in una casella libera il proprio simbolo, scelto tra una croce e un cerchio. Il giocatore che compone una schiera di tre propri simboli su una riga, una colonna o una diagonale, vince la partita. Il programma deve disegnare la scacchiera, chiedere all'utente le coordinate del suo prossimo simbolo, cambiare il giocatore di turno dopo ogni mossa e decretare il vincitore.

X		O
	X	O
O	O	X

[P6.28]

Esercizio 5 – *il codice Python*

```
def main():
    # Construct a new empty board.
    board = [[" "] * 3, [" "] * 3, [" "] * 3]

    # Keep making moves until a player has won.
    turn = "X"
    gameOver = False
    while not gameOver:
        if turn == "X":
            turn = "O"
        else:
            turn = "X"
        takeTurn(board, turn)
        gameOver = gameWon(board, turn) or isFull(board)

    # Display the final game board and the winner.
    drawBoard(board)
    if gameWon(board, turn):
        print("Player", turn, "won!")
    else:
        print("It was a tie.")
```

Esercizio 5 – *il codice Python*

```python
## Process the turn for one player.
#  @param board the game board to work with
#  @param mark the symbol used for the player
#
def takeTurn(board, mark):
    # Display the board.
    drawBoard(board)

    # Read the user's move, ensuring that it is legal.
    print("Player %s, make your move: " % mark)
    row = int(input("  row: "))
    col = int(input("  col: "))
    while board[row][col] != " ":
        print("That wasn't a valid move, try again: ")
        row = int(input("  row: "))
        col = int(input("  col: "))

    # Update the board with the mark.
    board[row][col] = mark
```

```python
def drawBoard(board):
    """
    Draw the game board

    :param board: the game board to display
    """
    print("   0   1   2")
    for i in range(0, 2):
        print("%s  %-2s| %-2s| %-2s" % (i, board[i][0], board[i][1], board[i][2]))
        print("  ---+---+---")
    print("2  %-2s| %-2s| %-2s" % (board[2][0], board[2][1], board[2][2]))


def gameWon(board, mark):
    """
    Determine if a player has won the game

    :param board: the board to check for a winner
    :param mark: the mark to check for winning
    :return: True if the game was won with the given mark, False otherwise
    """
    # Check the rows and columns for a winner.
    for i in range(0, 3):
        if board[i][0] == mark and board[i][1] == mark and board[i][2] == mark:
            return True
        if board[0][i] == mark and board[1][i] == mark and board[2][i] == mark:
            return True
    # Check the diagonals for a winner.
    if board[0][0] == mark and board[1][1] == mark and board[2][2] == mark:
        return True
    if board[2][0] == mark and board[1][1] == mark and board[0][2] == mark:
        return True
    return False
```

Esercizio 5 – *il codice Python*

```python
def isFull(board):
    """
    Determine if the board is full

    :param board: the board to check
    :return: True if the board is full, False otherwise
    """
    for i in range(0, 3):
        for j in range(0, 3):
            if board[i][j] == " ":
                return False
    return True


# Call the main function.
main()
```

Esercizio 5 – *il codice Python*

```
71  ## Controlla se la scacchiera sia piena o meno
72  #  @param board la scacchiera da controllare
73  #  @return True se la scacchiera è piena, altrimenti False
74  #
75  def isFull(board) :
76      for i in range(0, 3) :
77          for j in range(0, 3) :
78              if board[i][j] == " " :
79                  return False
80      return True
81
82  # Invoca la funzione main
83  main()
```