Relazione esonero

Approccio TopDown ricorsivo senza memorizzazione

L'idea è quella di utilizzare due funzioni, una funzione contaPercorsi() e la funzione ricorsione().

La prima funzione prende in input due interi n e k, che rappresentano la dimensione della matrice quadrata su cui calcolare i percorsi e il numero di svolte massime consentite durante il percorso tra la cella di partenza e quella di destinazione. Contapercorsi(n,k) controlla il verificarsi di diversi casi base:

- N<0: indica che la matrice immaginaria su cui calcolare i percorsi abbia una dimensione negativa, visto che la matrice deve avere una dimensione maggiore o uguale a 0, ritornerà -1,
- K<0: indica che il numero di svolte consentito è negativo, visto che il numero di svolte potrà essere al minimo 0, ritornerà -1,
- N==0: indica cha la matrice immaginaria su cui calcolare i percorsi abbia una dimensione 0x0, questo significa che non esisteranno percorsi possibili, poiché non esiste ne una cella di partenza ne una di destinazione, ritornerà 0,
- N==1: indica che la matrice immaginaria su cui calcolare i percorsi abbia una dimensione 1x1, questo significa che la cella di partenza e destinazione sarà la stessa e sarà possibile un solo percorso possibile che richiede zero svolte, ritornerà 1,
- K==0: non avendo sfruttato gli altri case base, significa che la matrice immaginaria su cui calcolare i percorsi abbia una dimensione di almeno 2x2 fino ad nxn, richiedendo così almeno una svolta, ma avendo a disposizione nessuna svolta, ritornerà 0.

Dopo aver verificato i casi base si decrementa n, poiché se abbiamo ricevuto in input n= 4, il calcolo dei percorsi deve avvenire su una matrice di dimensione 4x4, ma poiché gli indici delle matrici partono da 0, se usassimo n avremmo una matrice 5x5, che "sballerebbe" il calcolo dei percorsi, quindi si decrementa n di uno, avendo così effettivamente una matrice 4x4 con indici che vanno da 0 a 3.

Si richiama la funzione ricorsione() 2 volte a cui si passa la prima volta (n-1, n, k, True) e la seconda volta (n, n-1, k, False), andando così prima a ricercare le soluzioni che possono partire dalla cella sopra l'ultima a destra in fondo e poi quella a sinistra sempre dell'ultima cella, sarà la somma dei valori ritornati da queste chiamate a dare il risultato finale.

La funzione ricorsione() prende in input quattro parametri x, y, k, di.

X e Y rappresentano la cella immaginaria in una matrice nxn, indicando riga e colonna, k indica il numero di svolte ancora disponibili per arrivare alla destinazione e di indica la direzione che il percorso sta seguendo, quindi se si sta scorrendo una riga(False) o una colonna(True).

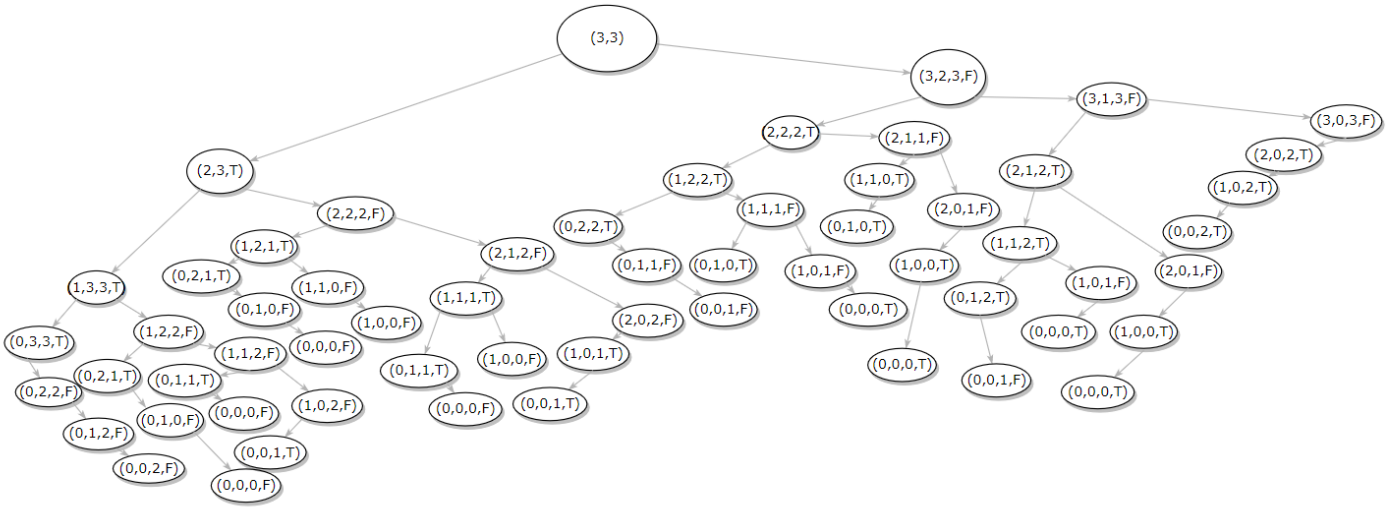
La funzione ricorsione() controlla se x ed y sono uguali a 0 allora ritorna 1, indicando così che si è trovato un percorso poiché si è arrivati alla destinazione, controlla poi se la direzione è True, significa che si sta viaggiando lungo una colonna, si controlla se andando verso l'alto si rimane all'interno della "nostra matrice" se è così si effettua una chiamata ricorsiva decrementando di 1 il numero di riga, continuando lungo la colonna, quando questo caso non è più verificato si controlla se andando verso sinistra si rimane all'interno della "matrice" e se sono rimaste delle svolte disponibili (se k-1 >= 0) , se il caso è verificato allora si aggiungono alle soluzioni precedentemente trovate quelle che si trovano richiamando la funzione ricorsiva andando verso sinistra (y-1), decrementando le svolte possibili (k-1) e negando di, indicando così che da quel momento ci si sta spostando lungo una riga, infine si ritorna la variabile w, precedentemente inizializzata a 0, che contiene il numero di percorsi ottenuti dalle chiamate appena indicate.

Dopo di che si affronta il caso in cui di sia False e quindi si sta viaggiando lungo una riga, si controlla se andando verso l'alto si rimane all'interno della nostra matrice e se le svolte rimanenti sarebbero almeno zero (visto che stavamo andando su una riga e ora ci stiamo spostando verso l'alto, abbiamo quindi effettuato una svolta), richiamiamo la funzione ricorsiva decrementando di 1 il numero di riga (x) e il numero di svolte e neghiamo la direzione, indicando che ora stiamo procedendo lungo una colonna, successivamente controlliamo che spostandoci verso sinistra non usciremmo dalla matrice, quindi richiamiamo la funzione ricorsiva decrementando il numero di colonna (y) e incrementando la variabile w del valore restituito dalla chiamata ricorsiva.

```python
def ricorsione(x,y,k,di):
  if (x == 0 and y == 0):
      return  1
  if (di):
      w = 0
      if(x-1 >= 0):
          w = ricorsione(x - 1, y, k, di )
      if( y-1 >= 0 and k-1 >= 0):
          w +=ricorsione(x, y - 1, k - 1, not di)
      return w
  else:
      w =0
      if(x-1 >= 0 and k-1 >= 0):
          w = ricorsione(x - 1, y, k - 1, not di)
      if(y-1 >=0):
         w += ricorsione(x, y - 1, k, di)
      return w

def contaPercorsi(n,k):
    if(n <0):
         return -1
    if(k<0):
        return -1
    if ( n == 0):
       return 0
    if ( n == 1):
        return 1
    if( k == 0 ):
        return 0
    n-=1
    return ricorsione(n-1, n, k, True) + ricorsione(n, n-1, k, False)
```

L'algoritmo proposto ha una complessità esponenziale poiché ricalcolerà gli stessi sotto problemi anche se già affrontati, non avendo modo di salvare e recuperare successivamente le informazioni per capire se ha già percorso un determinato cammino e ricalcolarlo sarebbe uno spreco di chiamate ricorsive. Possiamo vedere dall'albero sottostante che indica le varie chiamate ricorsive con nodi (x,y,k,di) (riga, colonna, svolte rimanenti, direzione) con n =4 e k=3.



L'algoritmo scenderà prima a sinistra e poi a destra

Approccio TopDown ricorsivo con memorizzazione

Per risolvere il problema dell'overlapping di problemi che viene illustrato dal disegno sopra.

L'algoritmo progettato si compone di 2 funzioni contaPercorsi() e trovaPercorsi ().

La funzione contaPercorsi() riceverà in input 2 interi n e k come la funzione contaPercorsi() proposta sopra, ed andrà a verificare i medesimi casi base già descritti in precedenza, questa volta però andrà anche a inizializzare la matrice mat che sarà una matrice di dimensione n*n*(k+1)*2, 2 per le possibili direzioni da seguire, k+1 per il numero di svolte compiute e n ed n per costruire una matrice quadrata n*2. La matrice mat sarà tutta inizializzata a 0 ed n decrementato di 1 in modo da non causare un errore relativo all'indice quando la seconda funzione cercherà di leggere i valori salvati nella matrice, se n=4 matrice avrà indici da 0 a 3.

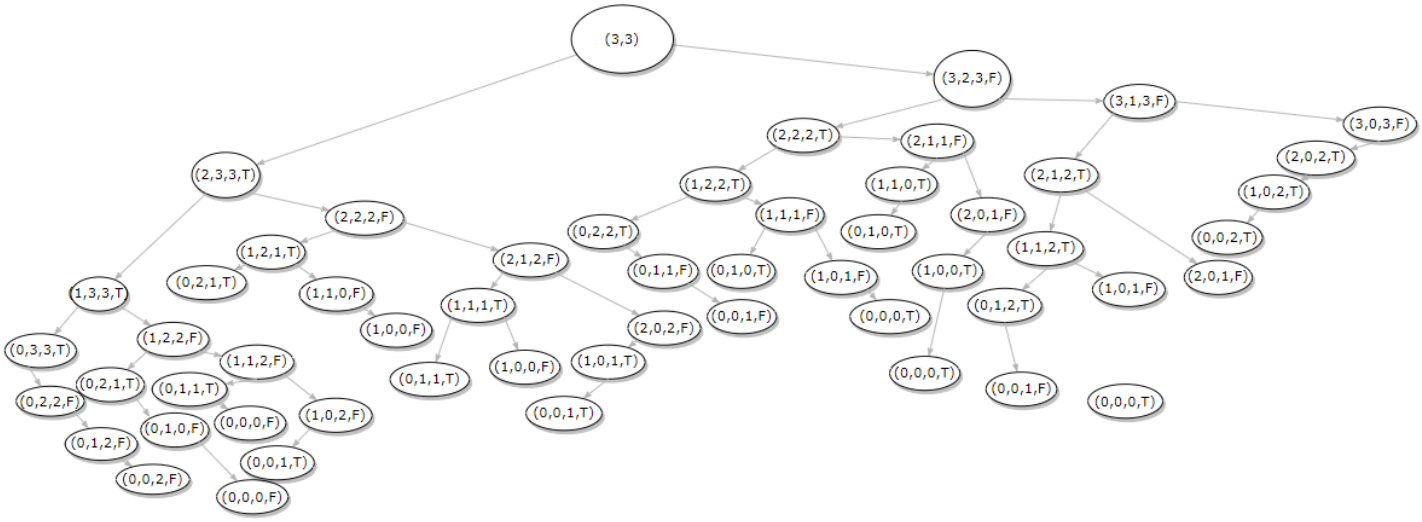
Dopo di che verrà richiamata per due volte la seconda funzione trovaPercorsi(), a cui verranno passati i parametri (n-1, n, k, True, mat) e la seconda volta (n, n-1, k, False, mat) e ritornerà la somma dei risultati ritornati dalla funzione ricorsiva.

La funzione trovaPercorsi () riceverà in input quindi x,y che sono il numero di righe e colonne, k il numero di svolte rimanenti, di che rappresenta la direzione presa e la matrice creata prima.

La funzione per prima cosa andrà a vedere se la cella mat[x][y][k][di] è diversa da 0, cioè se almeno un percorso con quelle stesse condizioni (in numero di svolte, direzione), ritornando così il valore della cella potremo sapere il numero di percorsi già trovati e terminare quel "ramo di ricorsione". Se mat[x][y][k][di] era uguale a zero allora si verifica se x ed y siano uguali a 0, in caso affermativo significa che abbiamo trovato un percorso e impostiamo la cella uguale ad 1.

Dopo di che verifichiamo che direzione stiamo seguendo attraverso il booleano "di" e se andando verso l'alto non usciamo dalla matrice, quindi finendo in un indice negativo, settiamo la cella attuale mat[x][y][k][di] uguale al valore ritornato dalla chiamata ricorsiva alla funzione trovaPercorsi(x-1, y, k, di, mat), dopo di che verifichiamo il caso in cui spostandoci a sinistra rimaniamo nella matrice e che effettuando questa futura svolta non si superi il numero di svolte consentite, in caso queste condizioni siano verificate allora incrementiamo la cella mat[x][y][k][di] del valore ritornato dalla chiamata ricorsiva alla funzione trovaPercorsi(x, y-1 ,k-1, not di, mat). Dopo queste verifiche ed assegnazioni/incrementi ritorniamo la cella in cui ci troviamo mat[x][y][k][di].

Se il booleano "di" risulterà False allora eseguiremo l'else in cui andremo a controllare se ci è ancora consentito spostarci verso l'alto e di effettuare una svolta, in caso affermativo assegniamo alla cella mat[x][y][k][di] il valore ritornato dalla chiamata ricorsiva trovaPercorsi(x-1, y, k-1, not di, mat), dopo di che andiamo a verificare se ci è consentito spostarci verso sinistra, in caso possiamo incrementeremo la cella mat[x][y][k][di] del valore ritornato dalla chiamata ricorsiva trovaPercorsi(x, y-1, k, di, mat). Dopo queste ultime operazioni ritorneremo il la cella mat[x][y][k][di].



Un esempio della ricorsione con la memorizzazione, ovviamente più la dimensione della matrice e il numero di svolte crescono maggiore saranno i "rami tagliati della ricorsione" a confronto con la versione non memorizzata, qui possiamo notare ad esempio la mancanza del ramo dopo (0, 2, 1, T) o dopo (1, 0, 1, F) a dx.

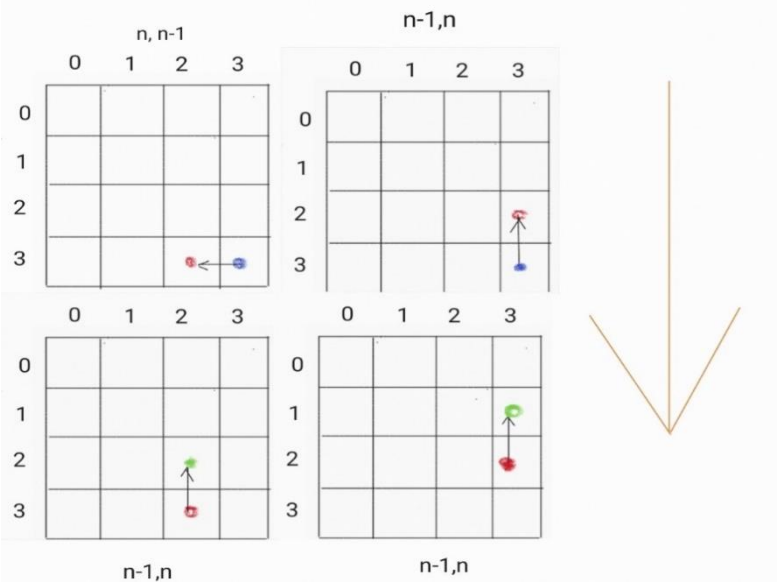
Codice dell'algoritmo che sfrutta la memorizzazione

```
def trovaPercorsi(x, y, k, di,mat):
    if (mat[x][y][k][di] != 0):
        return mat[x][y][k][di]
    if (x == 0 and y == 0):
        mat[x][y][k][di]= 1
    if (di):
        if(x-1 >= 0):
            mat[x][y][k][di] = trovaPercorsi(x - 1, y, k, di, mat)
        if( y-1 >= 0 and k-1 >= 0):
             mat[x][y][k][di] +=trovaPercorsi(x, y - 1, k - 1, not di, mat)
        return mat[x][y][k][di]

    else:
        if(x-1 >= 0 and k-1 >= 0):
            mat[x][y][k][di] = trovaPercorsi(x - 1, y, k - 1, not di, mat)
        if(y-1 >=0):
            mat[x][y][k][di] += trovaPercorsi(x, y - 1, k, di, mat)
        return mat[x][y][k][di]

def contaPercorsi(n, k):
    if(n <0):
         return -1
    if(k<0):
        return -1
    if ( n == 0):
       return 0
    if ( n == 1):
        return 1
    if( k == 0 ):
        return 0
    mat = [[[[0 for _ in range(2)] for _ in range(k+1)] for _ in range(n)] for _ in range(n)]
    n-=1
    return trovaPercorsi(n - 1, n, k, True, mat) +trovaPercorsi(n, n - 1, k, False, mat)
```

Esempio dell'inizio di esecuzione dell'algoritmo



La soluzione sarà data dalla somma delle celle:

mat[n-1][n][k][True] + mat[n][n-1][k][False]

L'algoritmo ha un costo di Θ (n*n*(k+1)*2),

per inizializzare la matrice, che semplificato

diventa Θ (n^2*k), per ogni quadrupla (x,y,k,di) verrà poi

calcolato il suo valore una volta e quindi il costo della

funzione ricorsiva sarà O(n^2*k),

avendo così un costo finale di O(n^2*k)

Possiamo calcolare i valori della tabella per righe e per colonne crescenti in base alla seguente regola:

Per questioni di implementazione nella versione BottomUp si è usato ‘z’ per ‘k’ e ‘w’ per ‘di’

mat[x][y][z][w] =

se x == 0, y == 0	mat[0][0][z][w] = 1
se w == True, x-1>=0, y-1>=0, z-1>=0	mat[x-1][y][z][1] + mat[x][y-1][z-1][0]
se w == True, x-1>= 0, y-1<0	mat[x-1][y][z][1]
se w == True, x-1<0, y-1>=0, z-1>=0	mat[x][y][z][w] + mat[x][y-1][z-1][0]
se w == False, x-1>=0, y-1>=0, z-1>=0	mat[x-1][y][z-1][1] + mat[x][y-1][z][0]
se w == False, x-1>=0, y-1<0, z-1>=0	mat[x-1][y][z-1][1]
se w == False, x-1<0, y-1>=0	mat[x][y][z][w] + mat[x][y-1][z][0]

Approccio BottomUp con memorizzazione

La versione BottomUp dell’algoritmo si forma sempre di due funzioni, una funzione ausiliaria contaPercorsi() e una che svolge i calcoli trovaPercorsi().

La funzione contaPercorsi(), che riceve in input due interi n, k che rappresentano la dimensione della matrice e il numero di svolte, verifica i vari casi base, e costruisce una matrice “mat” di dimensione n*n*(k+1)*2.

Dopo di che richiama la funzione che effettua la ricerca dei percorsi passandole (n-1, k, mat).

La funzione trovaPercorsi() esegue 4 for annidati, che scorrono la matrice, in questa versione si partirà dagli indici (0,0) al contrario delle altre due versioni che partivano da (n-1,n-1), cioè dalla cella in basso a destra . I for avranno range:

- 1’ for esterno: da 0 ad n+1,
- 2’ for annidato: da 0 ad n+1,
- 3’ for annidato: da 0 ad k+1,
- 4’ for annidato: da 0 a 2;

Nel quarto ciclo for vengono eseguite i vari controlli e assegnazioni/incrementi.

Se ci si trova nella cella (0,0) la cella mat[0][0][z][w] viene inizializzata ad 1, se invece questo caso non è verificato si controlla se w == True e decrementando di uno x e y non si esce dai margini della matrice e se ci sono ancora svolte disponibili (z-1>=0) allora la cella mat[x][y][z][w] avrà il valore dato dalla somma tra la cella mat[x-1][y][z][1] e

mat[x][y-1][z-1][0], cioè la cella sopra quella attuale e la cella a sinistra di quella attuale compiendo una svolta, poiché essendo w == True significa che ci stavamo spostando lungo una colonna e ora stiamo cambiando direzione andando a sinistra (y-1). Se invece w == True e possiamo controllare il valore della cella in alto ma non di quella a sinistra allora mat[x][y][z][w] prenderà il valore di mat[x-1][y][z][1], invece se w == True e non si può leggere quella in alto ma si può leggere quella a sinistra e si può compiere una svolta allora il valore della cella mat[x][y][z][w] sarà incrementato del valore della cella a sinistra mat[x][y-1][z-1][0].

Se w == False e come prima possiamo leggere il valore delle celle in alto e a sinistra e abbiamo ancora svolte disponibili allora il valore della cella mat[x][y][z][w] sarà uguale alla somma delle celle mat[x-1][y][z-1][1] e mat[x][y-1][z][0], invece rimanendo sempre con w == False abbiamo ancora due casi da analizzare, se è possibile leggere quella in alto ma non quella a sinistra e abbiamo ancora svolte disponibili allora il valore di mat[x][y][z][w] sarà uguale a quello di mat[x-1][y][z-1][1], l’ultimo caso rimasto da analizzare ora è quello se non si può analizzare quella in alto ma quella a sinistra è disponibile, allora il valore della cella mat[x][y][z][w] sarà incrementato del valore della cella mat[x][y-1][z][0], cioè quella a sinistra di quella analizzata. Infine la funzione una volta conclusi tutti i cicli for annidati, andrà a ritornare la somma dei valori contenuti nelle due celle mat[n-1][n][k][1] ed mat[n][n-1][k][0].

Codice dell'algoritmo BottomUp che sfrutta la memorizzazione

```
def trovaPercorsi(n, k, mat):
    for x in range(n+1):
        for y in range(n+1):
            for z in range(k+1):
                for w in range(2):
                        if x == 0 and y == 0: mat[0][0][z][w] = 1
                        elif w and x-1>= 0 and y-1>=0 and z-1>=0:
                            mat[x][y][z][w] = mat[x-1][y][z][1] + mat[x][y-1][z-1][0]
                        elif w and x-1>=0 and y-1<0:
                            mat[x][y][z][w] = mat[x-1][y][z][1]
                        elif w and x-1<0 and y-1>=0 and z-1>=0:
                            mat[x][y][z][w]+= mat[x][y-1][z-1][0]
                        elif w == False and x-1>=0 and y-1>=0 and z-1>=0:
                            mat[x][y][z][w] = mat[x-1][y][z-1][1] + mat[x][y-1][z][0]
                        elif w == False and x-1>=0 and y-1<0 and z-1>=0:
                            mat[x][y][z][w] = mat[x-1][y][z-1][1]
                        elif w == False and x-1<0 and y-1>=0:
                            mat[x][y][z][w]+= mat[x][y-1][z][0]

    return  mat[n-1][n][k][1] +  mat[n][n-1][k][0]
def contaPercorsi(n, k):

    if(n <0): return -1
    if(k<0): return -1
    if ( n == 1): return 1
    if( k == 0 ): return 0
    mat = [[[[0 for _ in range(2)] for _ in range(k+1)] for _ in range(n)] for _ in range(n)]
    return  trovaPercorsi(n-1, k, mat)
```

L'algoritmo ha un costo di Θ (n*n*(k+1)*), per inizializzare la matrice, che semplificato diventa Θ (n^2k) poichè le varie costanti vengono eliminate. Dopo di che abbiamo la funzione che ricerca i vari percorsi che eseguirà i vari for annidati, andando quindi ad aver un costo di Θ ((n+1)*(n+1)*(k+1)*2) che semplificato diventerà Θ (n*n*k).

Avendo così un costo totale dell'algoritmo che sarà O(n^2k).

Di seguito una serie di test che sono stati confrontati anche con altri studenti

N	K	PERCORSI
4	2	6
4	3	14
6	5	162
8	8	2716
12	2	22
12	6	15972
35	2	68