Count all possible paths in a grid n*n with at most k turns

Simone Nicolae Mija 1895592

Esonero progettazione algoritmi

1 Introduzione del problema

Vogliamo contare il numero di paths possibili con al più k turns (svolte) partendo dalla cella in alto a sinistra fino alla cella in basso a destra all'interno di una griglia n x n. Un percorso può essere costituito soltanto da movimenti verso destra e verso il basso. La specifica richiede un algoritmo in tempo O(n^2k).

2 L'algoritmo combinatorio

2.1 L'intuizione

Cerchiamo di risolvere il problema applicando un approccio combinatorio. Sappiamo che ci troviamo in una griglia n x n e vogliamo trovare tutti i possibili percorsi che hanno al più k turns. Possiamo notare che i vari paths possono essere rappresentati usando una stringa composta da R (che indica che ci siamo spostati verso destra) e D (che indica che ci siamo spostati verso il basso).

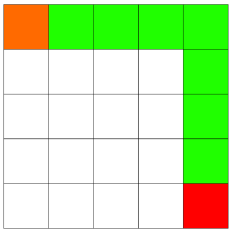


Figure 1: Il percorso è segnato in verde. Quest'ultimo può essere rappresentato come stringa RRRRDDDD

Banalmente, si nota che le dimensioni della stringa sono sempre le stesse, indipendentemente dal numero di turns, in quanto posso andare verso destra finchè non decido di eseguire un turn oppure arrivo alla fine della griglia; analogamente accade partendo dal basso, per simmetria. La dimostrazione può essere ulteriormente confermata applicando semplicemente la geometria: sapendo di trovarci all'interno di un quadrato, riusciamo a prenderci i casi limiti per raggiungere le due estremità, quindi sappiamo che raggiungere il punto $P_1(n-1, n-1)$ andando verso il basso fino alla fine della griglia, per poi andare verso destra fino al raggiungimento della destinazione è il caso con la lunghezza più grande. Quindi, la lunghezza della stringa sarà $(n-1)+(n-1)$ in quanto dovrò andare verso il basso $(n-1)$ volte, per poi andare a destra $(n-1)$ volte. Analogamente accade la stessa cosa partendo da destra, grazie alla simmetria. Il caso minimo invece è la diagonale, in quanto è la lunghezza minima per raggiungere l'estremita di un quadrato: andare in diagonale implica però che bisogna eseguire, sempre partendo verso il basso, una volta D e un'altra volta R. Questo processo va ripetuto tuttavia n-1 volte prima di raggiungere $P_1(n-1, n-1)$: si ottiene quindi $2(n-1)$. Otteniamo quindi il caso massimo $(n-1)+(n-1)$ e il caso minimo $2(n-1)$ che tuttavia sono uguali.

$$2(n-1) = (n-1)+(n-1) \tag{1}$$

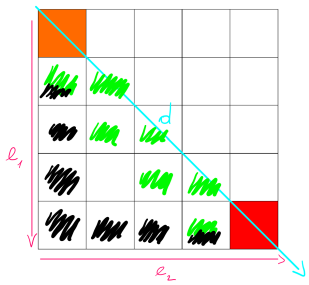


Figure 2: Il caso massimo è segnato in nero, il caso minimo in verde

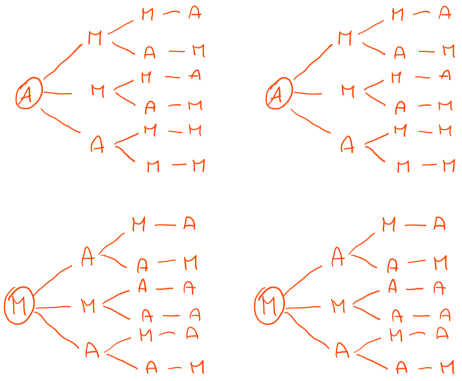
Banalmente, otteniamo anche che *il numero di R e il numero di D sono sempre uguali*, in quanto ci troviamo in un quadrato. Abbiamo anche ottenuto il valore minimo di k che è 1, e il valore massimo di k, ottenuto esguendo la diagonale della griglia.

$$k = [1; 2(n-1) - 1] \tag{2}$$

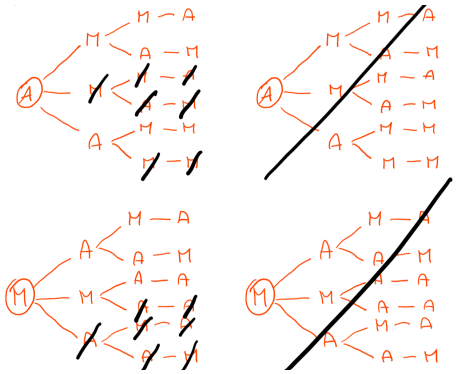
2.2 L'utilizzo della combinatoria

2.2.1 Ignoriamo il vincolo k turns

Proviamo a risolvere il problema, ignorando il vincolo dei k turns, quindi vogliamo trovare tutte le possibili stringhe che riusciamo ad ottenere. Dobbiamo quindi eseguire un anagramma di $lunghezzaStringa$ avente come unici caratteri R e D. Otteniamo quindi $(lunghezzaStringa)! = (2(n-1))!$. Tuttavia cosi facendo teniamo conto dell'ordine: bisogna levare quindi gli elementi ripetuti.



(a) Tenendo conto dell'ordine ottengo 24 stringhe possibili.



(b) Non tenendo conto dell'ordine, ottengo 6 stringhe possibili.

Figure 3: Un esempio di anagramma di "AMMA".

Il numero di tutti i possibili percorsi eseguibili diventa

$$\frac{(2(n-1))!}{(n-1)!(n-1)!} \tag{3}$$

ci servirà in futuro per l'algoritmo.

2.2.2 Consideriamo esattamente k turns

A questo punto proviamo a risolvere il problema, considerando anzichè al più k turns, esattamente k turns. Prendiamo la nostra forma di rappresentazione del percorso: la stringa. Fare k turns significa avere all'interno della stringa k+1 blocchi aventi lo stesso carattere ripetuto.

$$\mathscr{B}_0, \mathscr{B}_1, ..., \mathscr{B}_k \tag{4}$$

Definiamo a questo punto b_i come la lunghezza del blocco $\mathscr{B}_i$, e sappiamo che la somma dei blocchi è pari a $2(n-1)$

$$b_0 + b_1 + ... + b_k = 2(n-1) \tag{5}$$

A questo punto abbiamo due casi principali:

- il blocco $\mathscr{B}_0$ è composto da caratteri R $\Leftrightarrow$ la stringa inizia con R
- il blocco $\mathscr{B}_0$ è composto da caretteri D$\Leftrightarrow$ la stringa inizia con D

esaminiamo soltanto il primo caso.
Sappiamo quindi che i blocchi composti da un indice pari sono blocchi con all'interno caratteri R, mentre quelli avente indice dispari sono blocchi con all'interno caratteri B. Definiamo $m = \frac{k}{2}, m' = \frac{k-1}{2} \mid m, m' \in \mathbb{N} \Rightarrow m = \lfloor \frac{K}{2} \rfloor, m' = \lfloor \frac{K-1}{2} \rfloor$. Otteniamo quindi:

$$b_0 + b_2 + ... + b_{2m} = (n-1) \tag{6}$$

e

$$b_1 + b_3 + ... + b_{2m'+1} = (n-1) \tag{7}$$

Prendiamo soltanto la prima uguaglianza (6).
A questo punto dobbiamo soltanto contare tutte le possibili soluzioni che l'uguaglianza ha. Per fare ciò, basta utilizzare il primo teorema del Stars and bars theorem, che afferma che per ogni k'-tuple, con all'interno numeri naturali, la cui somma è pari a n', il numero di soluzioni possibili è $\binom{n'-1}{k'-1}$. Nel nostro caso abbiamo $n' = (n-1)$ e $k' = m+1$ (+1 in quanto contiamo anche l'indice 0). Otteniamo quindi

$$\binom{n-1-1}{m+1-1} = \binom{n-2}{\lfloor \frac{K}{2} \rfloor} \tag{8}$$

Mentre per la seconda uguaglianza (7) si ottiene

$$\binom{n-1-1}{m'+1-1} = \binom{n-2}{\lfloor \frac{K-1}{2} \rfloor} \tag{9}$$

2.2.3 Consideriamo al più k turns

A questo punto, conoscendo il numero di disposizioni di R caratteri all'interno dei blocchi pari, e il numero di disposizioni di D caratteri all'interno dei blocchi dispari, posso combinare le due disposizioni, ottenendo:

$$\binom{n-2}{\lfloor \frac{K}{2} \rfloor}\binom{n-2}{\lfloor \frac{K-1}{2} \rfloor} \tag{10}$$

Dobbiamo anche aggiungere il caso in cui si inizia con un blocco composto da D $\Leftrightarrow \mathscr{B}_0$ è un blocco composto da D. Questa volta dobbiamo unire le due disposizioni, ottenendo:

$$\binom{n-2}{\lfloor \frac{K}{2} \rfloor}\binom{n-2}{\lfloor \frac{K-1}{2} \rfloor} + \binom{n-2}{\lfloor \frac{K-1}{2} \rfloor}\binom{n-2}{\lfloor \frac{K}{2} \rfloor} = 2\binom{n-2}{\lfloor \frac{K}{2} \rfloor}\binom{n-2}{\lfloor \frac{K-1}{2} \rfloor} \tag{11}$$

Abbiamo calcolato il numero di percorsi all'interno di una griglia con esattamente k turns, ora torniamo all'origine del nostro problema, ovvero calcolare il numero di percorsi all'interno di una griglia con al più k turns. Avendo la soluzione per esattamente k turns, basta semplicemente mettere il tutto dentro una sommatoria.Il risultato finale quindi diventa:

$$\sum_{i=0}^{k} 2\binom{n-2}{\lfloor \frac{i}{2} \rfloor}\binom{n-2}{\lfloor \frac{i-1}{2} \rfloor} \tag{12}$$

2.3 L'algoritmo

Avendo la formula, ottenere un algoritmo che risolva il nostro problema è banale. Eseguiamo la formula (11) all'interno di un ciclo dove k aumenta ad ogni iterazione, ciò ci permette di simulare la sommatoria.

```
def countPaths0(n, k):
    finalCount = 0
    for i in range(1, k + 1):
        finalCount += 2 * \
                      (math.factorial(n - 2) / \
                       (math.factorial(math.floor(i / 2)) * math.factorial((n - 2) - math.floor(i / 2)))) * \
                      (math.factorial(n - 2) / \
                       (math.factorial(math.floor((i - 1) / 2)) * math.factorial((n - 2) - math.floor((i - 1) / 2))))
    print(round(finalCount))
```

Figure 4: Algoritmo si presta ad eseguire la formula (12)

Proviamo a calcolarci il tempo computazionale T_0 e lo spazio computazionale S_0.
Banalmente otteniamo $S_0 = \theta(1)$ in quanto utilizziamo memoria solo per le variabili, senza includere quindi nessun array le cui dimensioni dipendono da n. Per quanto riguarda il tempo computazionale, in ogni singola iterazione eseguiamo 6 volte math.factorial che viene eseguito in tempo $O(n)$. Il ciclo for viene eseguito in tempo $O(k)$. Otteniamo banalmente come costo computazionale $T_0(n) = O(n) * O(k) = O(n * k)$

Tuttavia, come scritto in (2), sappiamo che $K = [1; 2(n-1) - 1]$ quindi il costo finale sarà

$$T_0(n) = O(n * n) = O(n^2) \tag{13}$$

Otteniamo un tempo computazionale minore di $O(n^2 * k)$ ma possiamo fare di meglio.
Possiamo notare che per ogni iterazione viene volta svolta la funzione math.factorial, costringendoci a ricalcolare il fattoriale di $\lfloor \frac{i}{2} \rfloor$, di $\lfloor \frac{i-1}{2} \rfloor$ e due volte quello di $(n-2)$. Sappiamo tuttavia che il fattoriale di $n-2$ sarà sempre fattoriale più grande che dobbiamo calcolare, e allo stesso tempo che all'interno del calcolo di quest'ultimo, vi è anche il valore di $\lfloor \frac{i}{2} \rfloor$ e di $\lfloor \frac{i-1}{2} \rfloor$ per come è definito il fattoriale. A questo punto basta sfruttare la programmazione dinamica e creare un array dove in ogni posizione i viene memorizzato $i!$ basandosi sul valore contenuto nella posizione $i-1$.

```
def createFactorialArray(z):  # O(n)
    factorials = [1] * (z+1)
    for i in range(1, z+1):
        factorials[i] = factorials[i - 1] * i
    return factorials
```

Figure 5: L'algoritmo resistuisce l'array con all'interno i valori composti dal fattoriale di z

Utilizzando questo array, lo spazio computazione aumenta, diventando $O(n)$.
A questo punto, possiamo rieseguire la formula sfruttando l'array con all'interno i fattoriali appena creato

```python
def countPaths(n: int, k: int): #O(k)
    finalCount = 0
    factorials = createFactorialArray(n - 2)
    for i in range(1, k+1):
        finalCount += 2 *\
                      (int((factorials[n - 2]) / (factorials[math.floor(i / 2)] *
                      factorials[(n-2) - math.floor(i / 2)])) * \
                      (int((factorials[n - 2]) / (factorials[math.floor((i - 1) / 2)] *
                      factorials[(n-2) - math.floor((i - 1) / 2)]))
    return finalCount
```

Figure 6: L'algoritmo finale che permette di calcolare il numero totale di percorsi possibili

Otteniamo quindi che la funzione createFactorialArray(n-2) ha costo $O(n-2)$ mentre l'esecuzione completa del ciclo for costa $O(k)$ in quanto una singola iterazione esegue delle semplici costanti e il ciclo viene ripetuto k volte. Il costo finale diventa $T(n) = O(n-2) + O(k) = O(n+k)$
Ssappiamo che il valore massimo di k è pari a $2(n-1)-1$ per (2). Otteniamo finalmente

$$T(n) = O(n-2) + O(2n-3) = O(3n) = O(n) \tag{14}$$

Dobbiamo tuttavia inserire il caso in cui $k > 2(n-1)-1$, altrimenti il programma andrebbe in out of index, visto che $\frac{i}{2}$ diventerebbe più grande di $n-2$ e non riuscirebbe a trovare l'ultimo/gli ultimi fattoriali. Cosa analoga anche per $\frac{i-1}{2}$.
Basta a questo punto sfruttare la formula (3), ovvero calcolarci tutti i possibili percorsi all'interno della griglia, indipendentemente dal numero di turns che abbiamo.

```python
def countPaths(n: int, k: int): #O(k)
    finalCount = 0
    if k>=2*n-3:
        return int(math.factorial(2*n-2) / (math.factorial(n-1) * math.factorial(n-1)))
    else:
        factorials = createFactorialArray(n - 2)
        for i in range(1, k+1):
            finalCount += 2 *\
                          (int((factorials[n - 2]) / (factorials[math.floor(i / 2)] *
                          factorials[(n-2) - math.floor(i / 2)])) * \
                          (int((factorials[n - 2]) / (factorials[math.floor((i - 1) / 2)] *
                          factorials[(n-2) - math.floor((i - 1) / 2)]))
    return finalCount
```

Figure 7: L'algoritmo finale che permette di calcolare il numero totale di percorsi possibili, includendo il caso $k > 2(n-1)-1$

2.4 Costi computazionali finali

2.5 Best Case

Notiamo subito che se rientriamo nella condizione $k >= 2n-3$ otteniamo comunque un algoritmo con il tempo computazionale $\Omega(3n) = \Omega(n)$. Tuttavia, lo spazio computazionale diminuisce, ottenendo spazio costante $\Omega(1)$

2.6 Worst Case and Average Case

Riguarda il blocco di codice dell'else. Otteniamo $O(n)$ sia per quanto riguarda il tempo che lo spazio computazionale.

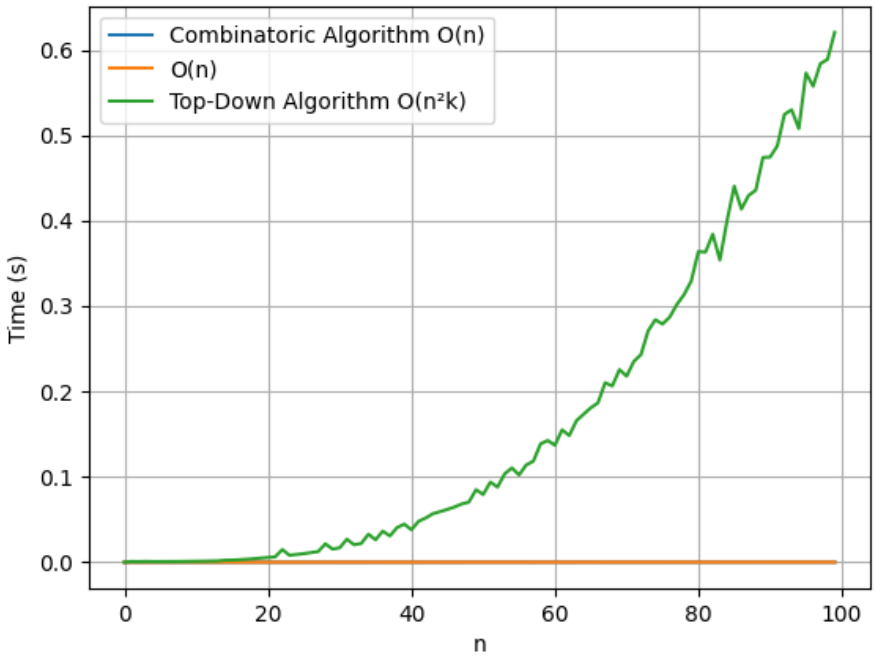


Figure 8: Differenza tra l'algoritmo combinatorio e un altro usando l'approccio top-down e la programmazione dinamica.

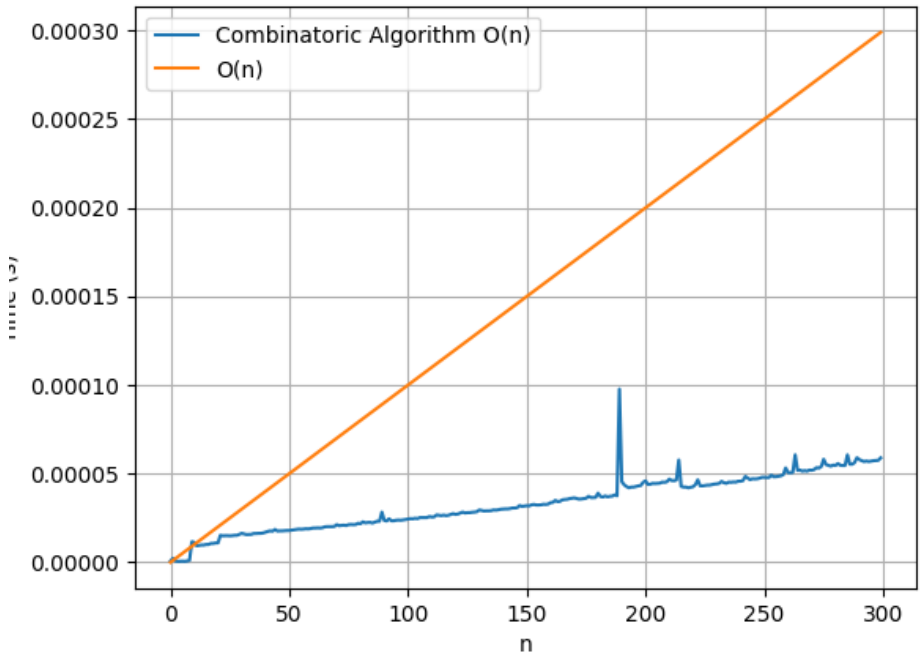


Figure 9: Grafico che sembrerebbe indicare che l'algoritmo si trovi in $O(n)$

3 L'inganno

Probabilmente andrà ben oltre la richiesta della traccia, ma penso tuttavia che andava approfondito ed inserito.

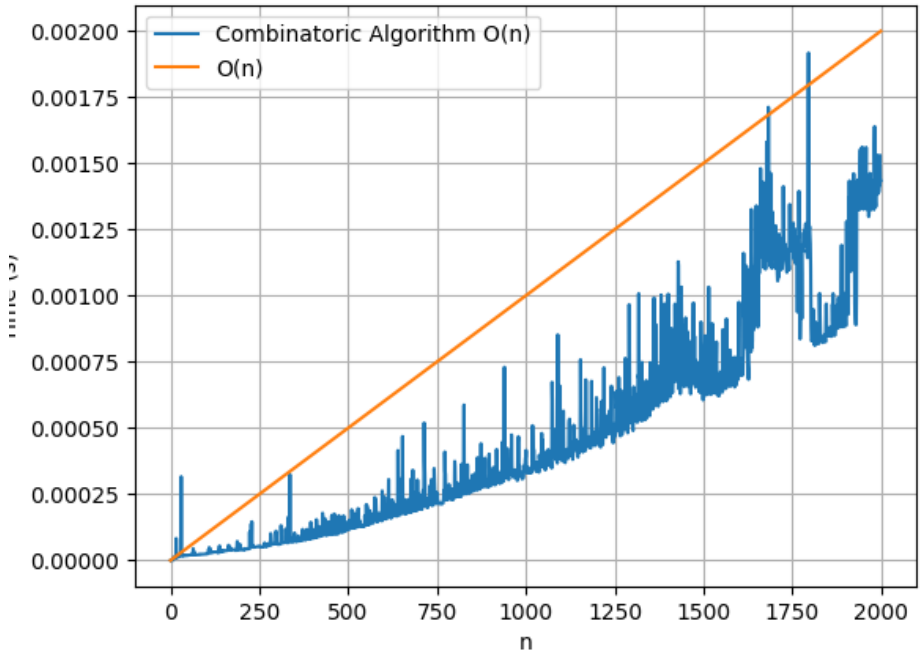


Figure 10: Forse in realtà non è $O(n)$

Cerchiamo di eseguire il fattoriale di 5, 15 e 100

$$5! = 120$$

$$15! = 1307674368000$$

$$\begin{aligned} 100! = 93326215443944152681699238856266700490715968264381621 \\ 468592963895217599993229915608941463976156518286253697 \\ 92082722375825118521091686400000000000000000000000 \end{aligned} \tag{15}$$

Notiamo fin da subito una problematica che non possiamo ignorare: la rappresentazione in bit dei numeri. Infatti

AumentaN $\Rightarrow$ Necessitiamo di più bit $\Rightarrow$ l'esecuzione della moltiplicazione avrà un costo maggiore e non sarà più una semplice costante

Ciò è dovuto dal fatto che non eseguiamo più dei valori con un numero di bit fissi (di solito 32 bit), ma ad ogni iterazione, ci serviranno sempre più bit.

3.1 Costo tempo computazionale

Sappiamo che eseguire una moltiplicazione, attraverso lo schoolbook multiplication algorithm (il classico algoritmo che viene anche insegnato a scuola per fare la moltiplicazione. E' usato da Python per numeri piccoli) ha un costo pari a $O(n'^2)$ dove n' è il numero di bit che ci servono per rappresentare il numero. Tuttavia, esistono algoritmi per calcolare la moltiplicazione tra due bit che vanno sotto il tempo quadratico. Python, quando i numeri da moltiplicare diventano molto grandi, anzichè utilizzare lo schoolbook multiplication algorithm, utilizza il Karatsuba Algorithm, che ha complessità pari

a $O(n'^{log_2 3})$. Quindi, per un numero decimale n'', per eseguire la moltiplicazione abbiamo come costo $O(\log^{\log_2 3} n'')$. Il calcolo della complessità per creare un fattoriale, ovvero la funzione `createFactorialArray()` cambia a questo punto e non sarà più

$$\sum_{i=1}^{n} O(1) = O(n)$$

ma bensì,sfruttando la proprietà dei logaritmi

$$\log(mn) = \log(m) + \log(n) \tag{16}$$

otteniamo

$$\sum_{i=1}^{n} O(\log^{log_2 3}(i)) = O(\log^{log_2 3} \prod_{i=1}^{n} i) = O(\log^{log_2 3}(n!)) \tag{17}$$

Prendiamo soltanto $O(\log(n!)$ e calcoliamoci esattamente la classe di appartenenza di $O(\log(n!))$. Per farlo, sfruttiamo l'asintotica e le proprietà dei logaritmi. Ricordiamoci la proprietà dei logaritmi (16) e la definizione di O(g(n))

$$O(g(n)) = \{f(n) : \exists\, c > 0,\ n_0 \geq 0 \mid f(n) \leq cg(n)\ \forall\ n \geq n_0\}$$

Prendiamo $f(n) = \log(n!)$, $g(n) = n\log(n)$, $c = 1, n_0 = 1$ e come ipotesi che $O(\log(n!)) \in O(n\log(n))$

$$\log(n!) = \log(n) + \log(n-1) + ... + \log(1)$$
$$n\log(n) = \log(n) + \log(n) + ... + \log(n)$$

a qusto punto, banalmente

$$\log(n) + \log(n-1) + ... + \log(1) \leq \log(n) + \log(n) + ... + \log(n) \quad \forall n \geq n_0$$

La dimostrazione può essere effettuata anche attraverso l'uso della Stirling Approximation, che conferma a sua volta la dimostrazione appena svolta. Sapendo che $O(log(n!)) = O(n\log(n))$, posso a questo punto effettuare l'elevazione a $log_2(3)$ e otteniamo come costo computazionale finale della funzione `createFactorialArray()`

$$T(n) = O(n^{\log_2 3}\log^{\log_2 3}(n)) \approx O(n^{1.58}\log^{1.58}(n)) \tag{18}$$

Analogamente, anche la funzione `countPaths()` avrà lo stesso costo computazionale, quindi si ottiene, per asintotica

$$T(n) = O(n^{1.58}\log^{1.58}(n))$$

3.2 Costo spazio computazionale

Per quanto riguarda il calcolo dello spazio computazionale, sappiamo che nel best case scenario ($k >= 2n-3$) utilizziamo $log_2(n!)$ bits per rappresentare il fattoriale. Quindi avremo

$$S(n) = \Omega(log(n!) = \Omega(nlog(n))$$

Cosa accade invece nel worst case e nell'avarage case? Sappiamo che avendo n_0 dati rappresentati da 32 bit in memoria, lo spazio computazionale assume il valore di $O(32n_0) = O(n_0)$. Nel nostro caso però, come già espresso, non abbiamo un numero costante di bit: arrivati ad un certo punto, ad ogni iterazione il numero di bit necessari cambierà. Il calcolo quindi non sarà

$$S(n) = \sum_{i=1}^{n} O(1) = O(n)$$

ma bensì

$$S(n) = O(\log(n!) + \log((n-1)!) + ... + \log(1!)) = O(\sum_{i=1}^{n} log(i!)) \tag{19}$$

Sappiamo tuttavia che $O(log(i!)) = O(nlog(n))$.

$$O(\sum_{i=1}^{n} log(i!)) = O(\sum_{i=1}^{n} i \cdot \log(i))$$

Possiamo approssimare la sommatoria con l'integrale $\int_1^{n+1} x \log(x)\, dx$

$$O(\sum_{i=1}^{n} i \cdot \log(i)) \approx O(\int_1^{n+1} x \log(x)\, dx)$$

Risolvendo l'integrale otteniamo

$$O(\int_1^{n+1} x \log(x)\, dx) = O(n^2 \log(n)) \tag{20}$$

Otteniamo quindi

$$S(n) = O(n^2 \log(n))$$

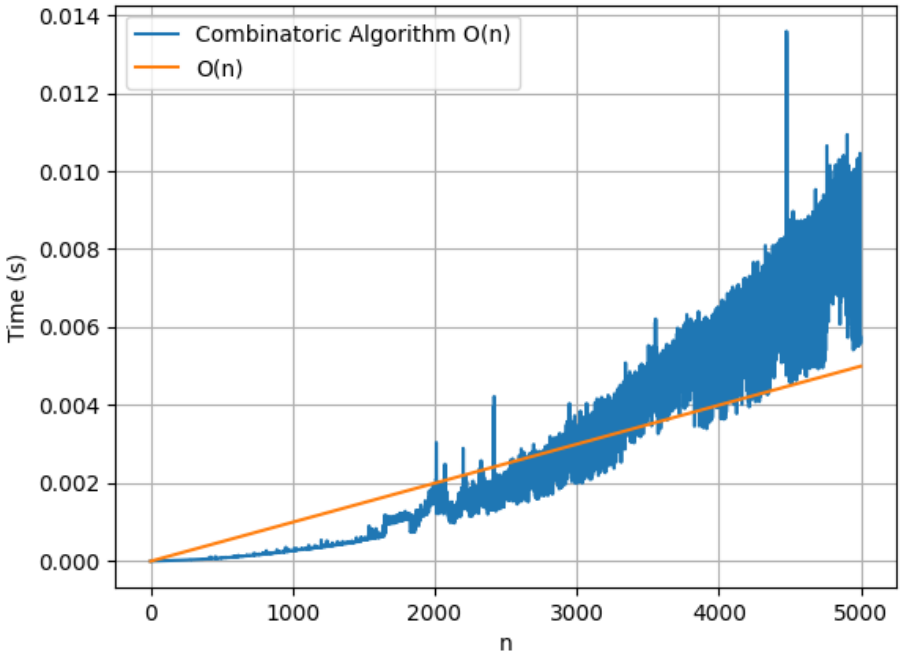


Figure 11: Si può notare il superamento di $O(n)$

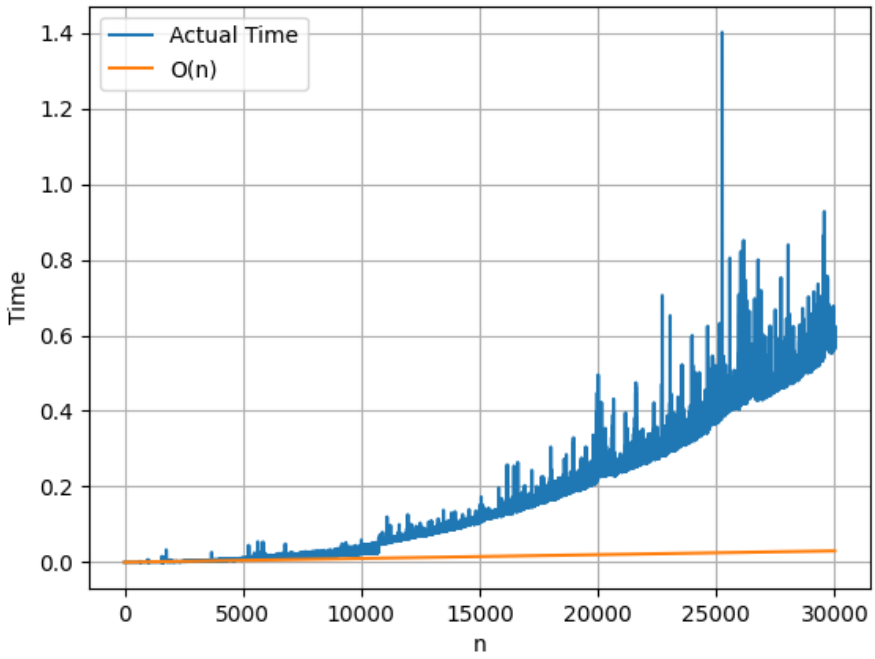


Figure 12: Si nota chiaramente che l'algoritmo non poteva essere in $O(n)$

4 Algoritmo Top-Down

L'algoritmo Top-Down si basa sullo sfruttamente la ricorsione per trovare tutti i possibili percorsi e man mano sommarmi quando eseguo il return delle chiamate ricorsive, sfruttando la programmazione dinamica per evitare l'overlapping di subproblems già risolti. Il costo dell'algoritmo è $O(n^2k)$ ed è stato usato per confrontarlo con l'algoritmo combinatorio. Il codice è commentato a dovere e pienamente comprensibile attraverso, appunto, i commenti

5 Uso del file .py

- `printPathsTD(n, k)`: printa il numero di percorsi possibili in una griglia nxn con al più k turns usando l'algoritmo Top-Down
- `printPathsCA(n, k)`: printa il numero di percorsi possibili in una griglia nxn con al più k turns usando l'algoritmo combinatorio
- `showChartCA(numberOfElement, k)`: mostra un grafico con una funzione $O(n)$ e il tempo impiegato dall'algoritmo combinatorio in base ad n
- `showChartWithTD(numberOfElement, k)`: mostra un grafico con una funzione $O(n)$, il tempo impiegato dall'algoritmo combinatorio in base ad n e il tempo impegato dall'agoritmo Top-Down in base ad n

Per eseguire i plot, bisogna avere la libreria `matplotlib`