RELAZIONE PER L'ESONERO DEL CORSO DI PROGETTAZIONE DI ALGORITMI

Di Scarselli Ilaria, matricola 1918975

Anno Accademico 2021/2022

1. Idee iniziali

Ragionando sul problema, mi sono venute in mente diverse idee. Inizialmente ho pensato di impostare l'esercizio provandolo a risolvere con i grafi in un modo molto simile al classico esercizio dell'acqua con i tre contenitori a capienza fissa da (ad esempio, ma dipende dall'istanza del problema che consideriamo) 4, 7 e 10 litri, rispettivamente. In questo esercizio si suppone di avere inizialmente i primi due contenitori pieni e di poter versare l'acqua da un contenitore a un altro fermandoci solo quando il contenitore sorgente risulti vuoto o quello destinazione risulti pieno. Vogliamo sapere se c'è una sequenza di versamenti che termina lasciando esattamente un numero tot di litri in uno dei primi due contenitori. Questo problema può essere rappresentato mediante l'uso dei grafi, in cui ogni nodo è una configurazione che descrive i possibili stati di riempimento dei tre contenitori (ci sono anche nodi che dalla configurazione iniziale non possono essere raggiunti). L'arco (u,v) indica che si può passare dalla configurazione descritta dal nodo u a quella del nodo v attraverso un'azione di versamento legale. A questo punto si riduce il problema a quello di trovare un cammino dal nodo iniziale a uno di quelli che soddisfano la richiesta. Analogamente, quello che avevo pensato di fare era di impostare la configurazione come (i,j,k), dove i indicava il numero di riga attuale, j il numero di colonna, k il numero di svolte che avevo fatto fino a quel momento, oppure di prendere le configurazioni che indicavano le coordinate delle caselle in cui effettuavo le svolte. Alla fine di tutto, se fossi arrivata alla cella voluta senza che k avesse superato il numero di svolte massime richieste dall'input, avrei potuto incrementare un contatore dei possibili percorsi. Tuttavia, in questo caso, la situazione sarebbe stata molto più complicata rispetto al caso precedente, perché il problema non richiede di verificare che esiste un percorso, ma richiede di contarne tutti i possibili; quindi, una volta finito un percorso, sarei dovuta tornare indietro per calcolare man mano tutti i possibili altri. Inoltre,

probabilmente mi sarebbe servito ricordare anche la direzione corrente in cui si stava procedendo e in più il numero dei nodi, che poteva essere anche molto alto, dipendeva ovviamente dall'n in input, portando a molti percorsi errati che se non gestiti adeguatamente avrebbero potuto complicare il problema.

Ragionando su questi fattori ho deciso di provare a optare per un altro approccio. Ho scelto di seguire, sviluppare e ottimizzarne due, uno che si basa più propriamente sulla programmazione dinamica e uno che si basa maggiormente sull'uso della combinatoria.

2. Programmazione dinamica

Due concetti fondamentali per la programmazione dinamica sono l'overlapping di problemi e la memoizzazione. Un problema che presenti l'overlapping, ovvero più sottoproblemi uguali che si ripetono e che quindi portano alla necessità di risolvere più volte la stessa cosa rendendo enormemente più elevato il costo computazionale, potrebbe essere semplificato dall'uso della memoizzazione, che permette man mano di salvare in tabelle le soluzioni dei sotto problemi già incontrati per renderle facilmente reperibili in caso di necessità.

2.1 Algoritmo con ricorsione semplice

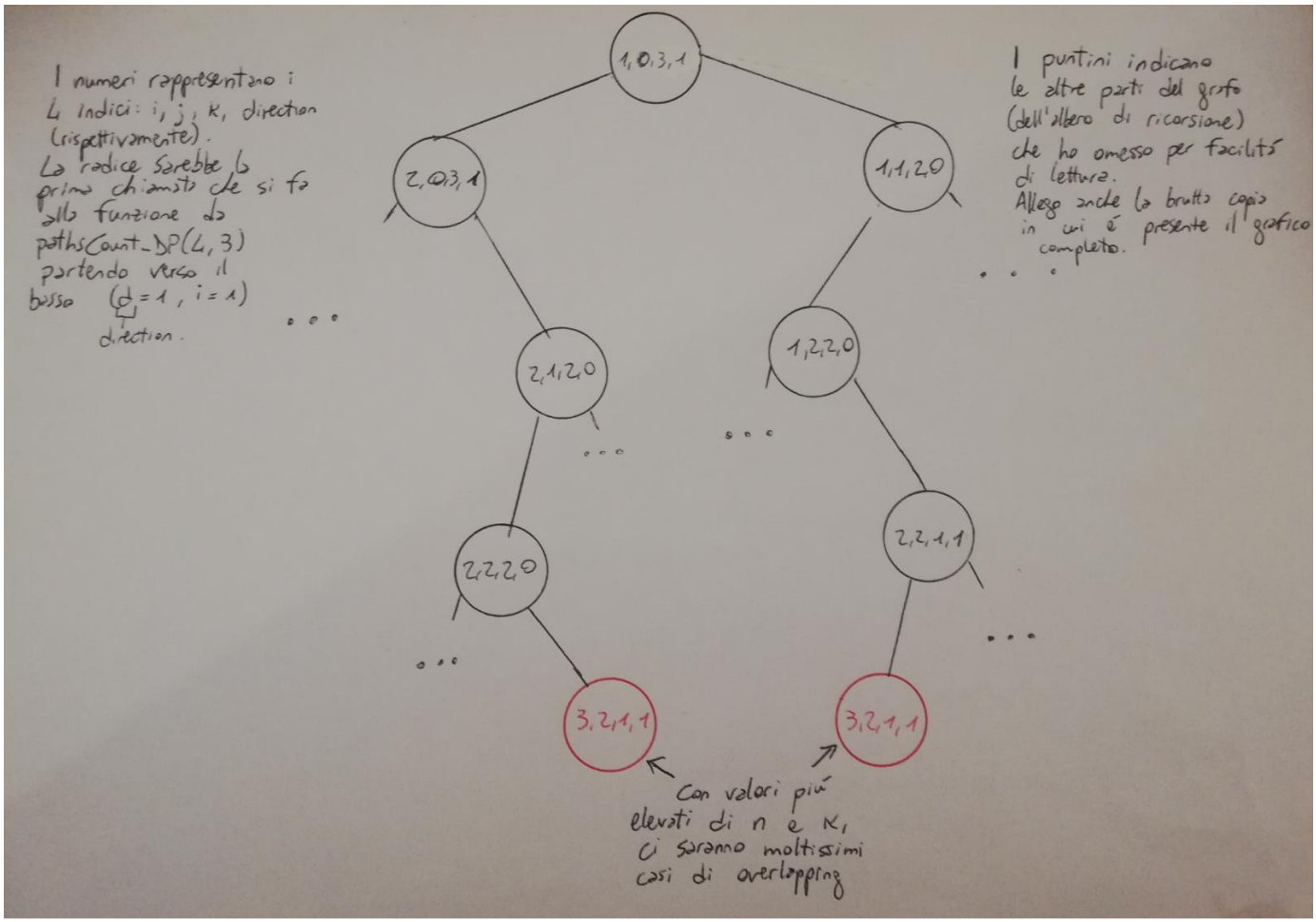
```
3   def pathNum(n, k):
4       if k == 0:
5           return 0
6       if k == 1:
7           return 2
8       if k > 2*n-3:
9           k = 2*n-3
10      return pathsForK(n, 1, 0, k, 1) + pathsForK(n, 0, 1, k, 0)
11
12
13  def pathsForK(n, i, j, k, direction):
14      if k == -1 or i >= n or j >= n:
15          return 0
16      if ((direction == 1 and j == n-1) or
17          (direction == 0 and i == n-1)):
18          return 1
19      if direction == 1:
20          return pathsForK(n, i + 1, j, k, direction) + pathsForK(n, i, j + 1, k - 1, 0)
21      else:
22          return pathsForK(n, i, j + 1, k, direction) + pathsForK(n, i + 1, j, k - 1, 1)
```

La funzione pathNum controlla i casi banali in cui k è uguale a zero (e in questo caso, supponendo di avere una matrice di lato almeno 2, non si potrebbe arrivare alla destinazione, muovendosi solo verso destra o verso il basso), a 1 (ci si muove prima completamente a destra fino al bordo e poi completamente verso il basso o viceversa) o quando è maggiore di 2*n -3, che è il numero di svolte massimo ottenibile seguendo la diagonale (in tal caso considero i k fino al massimo, altrimenti non troverei comunque niente di nuovo).

Inizialmente la mia idea era di usare pathsForK all'interno di un ciclo da 1 a k per calcolare i percorsi con esattamente k svolte e poi sommarli man mano e restituire infine il risultato, ma alla fine si può tranquillamente evitare il ciclo aggiungendo un paio di controlli all'interno della funzione pathsForK e ho optato per questa opzione. Quindi, alla fine di pathNum restituisco il numero totale di percorsi con al massimo k svolte sommando i valori ottenuti da due chiamate ricorsive, una che parte andando verso il basso e l'altra che parte andando verso destra.

In pathsForK, i rappresenta il numero della riga attuale, j il numero di colonna e ho aggiunto una variabile direction che indica la direzione in cui sto procedendo attualmente (1 se sto andando verso il basso, 0 verso destra). Calcolo i vari casi base: quello in cui mi trovo con un k non valido oppure in una cella non valida fuori dalla griglia n x n, nel qual caso ritorno 0 per indicare che non c'è un percorso valido da aggiungere; quello in cui mi trovo con la direzione attuale verso destra e mi trovo nell' ultima riga, oppure quello in cui sto andando verso il basso e mi trovo nell'ultima colonna. In questi ultimi casi riuscirò sicuramente ad arrivare senza dover svoltare alla cella finale e perciò ritorno 1. Se non mi ritrovo in questi casi base, ho altre due possibilità: sia che mi sto muovendo verso destra, che verso il basso, posso decidere di continuare per quella direzione o di svoltare. Quindi, sia per il ramo if che per l'else, faccio partire due chiamate ricorsive per valutare entrambe le scelte, incrementando di conseguenza il numero della riga o della colonna attuale in base a dove andrò.

Il costo computazionale di questo algoritmo è esponenziale, perché si ritrova ad affrontare lo stesso sottoproblema varie volte. L'overlapping è visibile anche solo osservando una piccola porzione di questo esempio:



2.2 Algoritmo ricorsivo con memoizzazione in tabella n x n x k x direction

Come detto in precedenza i problemi di overlapping possono essere risolti memorizzando i risultati in una tabella, che in questo caso è a 4 dimensioni, una per ogni parametro che voglio tenere sotto controllo e per cui voglio differenziare i casi.

```python
# Non considero il caso in cui la tabella è di 1 x 1,
# in quanto la casella di partenza e quella di destinazione
# combaciano
def pathsCount_DP(n, k):
    if k == 0:
        return 0
    if k == 1:
        return 2
    if k > 2*n-3:
        k = 2*n-3
    M = [[[[-1 for _ in range(2)]
            for _ in range(k)]
            for _ in range(n)]
            for _ in range(n)]
    return PathsForK_DP(n, 1, 0, k, 1, M) + PathsForK_DP(n, 0, 1, k, 0, M)


def PathsForK_DP(n, i, j, k, direction, M):
    if i >= n or j >= n:
        return 0
    if i == n-1 and j == n-1:
        return 1
    if (k == 0):
        if ((direction == 1 and j == n-1) or
            (direction == 0 and i == n-1)):
            return 1
        else:
            return 0
    if M[i][j][k-1][direction] != -1:
        return M[i][j][k-1][direction]
    if direction == 1:
        M[i][j][k-1][direction] = (PathsForK_DP(n, i + 1, j, k, direction, M) +
                                   PathsForK_DP(n, i, j + 1, k - 1, 0, M))
        return M[i][j][k-1][direction]
    else:
        M[i][j][k-1][direction] = (PathsForK_DP(n, i, j + 1, k, direction, M) +
                                   PathsForK_DP(n, i + 1, j, k - 1, 1, M))
        return M[i][j][k-1][direction]
```

Quello che cambia nella prima funzione è che, oltre a valutare i casi base, creo e inizializzo una tabella a quattro dimensioni: un indice per le righe della griglia originaria, uno per le colonne (che sarà sempre n), uno per i k che abbiamo ancora a disposizione al momento e uno per la direzione attuale, in modo tale da andare poi a riempire la cella che descrive esattamente la situazione in cui mi trovo. Infine, in questa funzione, faccio le due chiamate alla funzione PathsForK_DP come prima, in cui l'unica differenza è che passo anche la tabella M come parametro.

Anche la funzione PathsForK_DP funziona in modo molto simile alla precedente. Modifico uno dei controlli: se k è uguale a zero, ma non ci troviamo né nella riga n-1, né nella colonna n-1, allora sarà impossibile raggiungere la destinazione e quindi ritorno direttamente 0. Se ho già calcolato il valore della cella della tabella M che descrive la

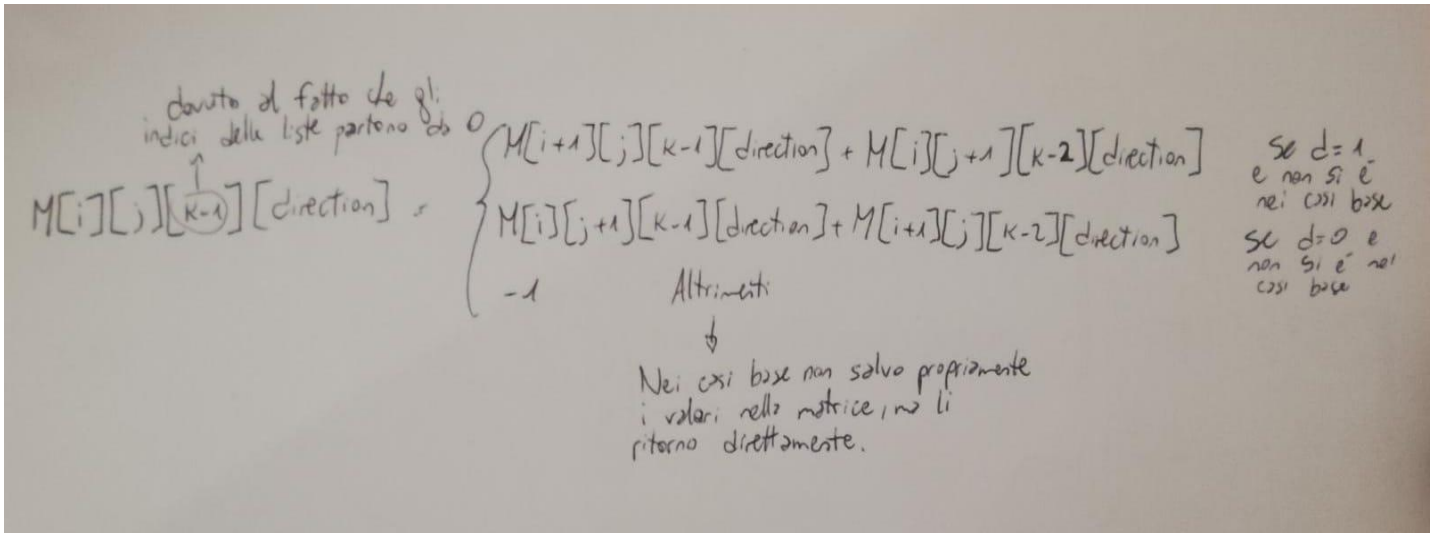
situazione attuale, allora posso non ricalcolarla e restituirla direttamente, altrimenti la calcolo in modo analogo a quanto fatto prima (con i due casi suddivisi in base alla direzione corrente) e poi salvo la somma delle due chiamate ricorsive nella cella attuale, per averla disponibile in futuro, prima di ritornarla.

Il costo asintotico è determinato dalla costruzione della tabella nella prima funzione, in quanto tutti gli altri costi di quella funzione sono costanti. Anche per quanto riguarda PathsForK_DP, non ci sono costi che eccedono quelli della creazione della tabella, in quanto ci sono solo costi costanti dovuti da controlli, operazioni e scritture o letture dalla matrice M, che dipendono da quante volte vengono effettuate le chiamate ricorsive. Tuttavia, possiamo notare che un elevato numero di caselle in M non sono raggiungibili dalla chiamata iniziale e le restanti hanno una sola operazione di scrittura più altre di lettura, ma appena si incontra una casella già scritta si termina quel ramo di ricorsione ritornando il valore e non si andranno a ricontrollare tutte le altre "sottostanti". Per cui, alla fine il costo è dato dai cicli per l'inizializzazione della tabella, n*n*k*direction, dove direction è solo 2, una costante. Quindi ottendiamo O(2*n^2k) = O(n^2k). Ma dato che la tabella deve essere necessariamente inizializzata tutta, sarà anche $\Omega(n^2k)$ che ci porta a $\theta(n^2k)$. Come osservazione finale, possiamo notare che k non può essere maggiore di 2n – 3, e quindi possiamo ottenere anche, come limite superiore, O(n^3) nel caso peggiore. A causa della tabella.

Sarebbero sicuramente possibili ottimizzazioni, come ad esempio quella di aggiungere un controllo per verificare se ci siamo muovendo verso destra e ci troviamo sull'ultima riga (o verso il basso nell'ultima colonna) anche per il caso in cui k non sia pari a 0, ma non diminuirebbero in maniera significativa i costi e i costi asintotici non cambierebbero.

Questo approccio (come anche il precedente) è Top-Down, in quanto si parte dal problema generale, si scompone in sottoproblemi minori attraverso l'uso della ricorsione per poi calcolare i risultati dei vari sottoproblemi e combinarli.

Infine, la relazione di ricorrenza che ha permesso di riempire la tabella è:



2.3 Algoritmo iterativo: idea

Ora che abbiamo la memoizzazione si potrebbe procedere all'approccio iterativo Bottom-Up, che consiste nel partire direttamente dai dettagli del problema, ovvero dalle foglie dell'albero di ricorsione visto in precedenza, e man mano costruire il risultato, consentendoci di liberarci della ricorsione. Questo si otterrebbe sfruttando la formula ricorsiva con cui è stata costruita la tabella in precedenza per riempire anche questa volta la tabella casella per casella, attraverso l'utilizzo di quattro cicli for annidati, uno per ogni indice. Se seguiamo l'approccio precedente, dobbiamo iniziare a riempire la tabella dalle ultime caselle (quelle che rappresentano l'ultima casella nella griglia di partenza) e risalire di volta in volta verso la casella iniziale. Al contrario di quanto visto prima, qui attraverso i cicli si riempie tutta la tabella. Il costo computazionale rimane comunque lo stesso, perché dominato dalla creazione della tabella, poi dal suo riempimento e da due letture per ogni sua casella (una per creare il valore della cella a sinistra e una per la cella in alto). Un'ottimizzazione possibile consisterebbe nel tenere memorizzata solo la parte di tabella che ci serve, in quanto man mano che risaliamo nelle righe, quelle non immediatamente più in basso non ci servono più. Tuttavia, questo costringerebbe comunque a scritture e riscritture e porterebbe ad aumentare il costo computazionale con costanti nascoste.

3. Approccio combinatorio

Alla base dell'idea per questo approccio ci sono due osservazioni:

a) Per qualsiasi percorso dal punto di partenza in alto a sinistra a quello di arrivo in basso a destra, si attraversano lo stesso numero di caselle e si fanno quindi lo stesso numero di mosse (2n - 2). Sia che ci si muove lungo la diagonale, che lungo i bordi, infatti, avendo solo due tipi di mosse a disposizione non si potrà tornare indietro verso l'alto o verso sinistra.
b) In qualsiasi percorso valido, si avranno lo stesso numero di mosse verso destra e verso il basso. Avendo un quadrato n x n, infatti, ci si deve spostare di n-1 caselle verso destra per arrivare al bordo destro e di n-1 caselle verso il basso.

Con queste due osservazioni posso descrivere tutti i possibili percorsi come una stringa composta da due caratteri che si ripetono: D (destra) e B (basso). Il problema ora può essere ridefinito come il trovare il numero di permutazioni (o meglio combinazioni, per essere più corretti, in quanto il problema può essere paragonato a un'istanza di quello delle stelle e barre) che soddisfano la condizione data (ovvero che il numero di svolte non deve superare un dato k). Si dovranno contare quindi tutti i vari modi leciti di disporre ad esempio il carattere D all'interno della stringa (e quindi conseguentemente anche il carattere B). C'è una svolta ogni volta che due simboli adiacenti sono diversi all'interno della stringa e questo crea K+1 blocchi di simboli identici.

Per calcolare il numero di stringhe con esattamente k svolte posso usare la formula:

$$2\binom{n-2}{\lfloor k/2 \rfloor}\binom{n-2}{\lfloor (k-1)/2 \rfloor}$$

Dove il 2 prima dei due coefficienti binomiali è dovuto al fatto che per simmetria ci sono lo stesso numero di stringhe sia iniziando con una mossa verso destra che con una verso il basso.

Per calcolare il numero di stringhe con al più k svolte posso utilizzare un ciclo che va da 1 a k e man mano mi somma i risultati ottenuti.

```python
def paths_comb(n, k):
    if k == 0:
        return 0
    if k == 1:
        return 2
    if k > 2*n-3:
        k = 2*n-3
    fact = [1 for _ in range(n+1)]
    # posso partire da 2 perché 0! = 1 e 1! = 1
    for x in range(2, n+1):
        fact[x] = fact[x - 1] * x
    perm = 0
    for i in range(1,k+1):
        perm += int(2 * fact[n-2] / (fact[i//2]*fact[n-2-i//2]) *
                    (fact[n-2] / (fact[(i-1)//2]*fact[n-2-(i-1)//2])))
    return perm
```

Lo spazio in memoria occupato in questo caso è dovuto a una lista di n elementi per calcolare i fattoriali e di una variabile per contare le varie "permutazioni" (in realtà come detto prima combinazioni). La lista contiene solo n elementi in quanto anche se k può essere 2n -3, quello che ci servirà nella formula è il fattoriale di k//2. Il costo è dunque $\theta(1) + \theta(n) + \theta(n) + \theta(k)$, ovvero il costo dei controlli e delle operazioni iniziali + quello della creazione e inizializzazione della lista + quello del calcolo dei fattoriali per ogni cella della lista (con operazioni che richiedono costo costante svolte per n volte) + quello per calcolare il numero di combinazioni nel ciclo per ogni k = $\theta(n + k) = \theta(n)$, dato che k è al massimo 2n-3. Sono possibili ottimizzazioni se consideriamo che se k è più grande di 2n-3, questo vuol dire che voglio semplicemente tutti i possibili percorsi ed esiste una formula diretta per la risoluzione di questo problema che mi consente di liberarmi del ciclo per k e di non essere costretta a tenere tutta la tabella memorizzata.
In un certo senso si può notare che anche in questo caso si fa uso della memoizzazione, salvando i vari fattoriali all'interno della matrice mono-dimensionale. Se non avessi salvato questi valori, li avrei dovuti calcolare da capo per ogni iterazione del ciclo per k, creando overlapping.