UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Algoritmi e strutture dati:

Homework 2

Anno Accademico 2023/2024

Salvatore Santoro M63001420
Andrea Russo M63001634

Problema 1

Traccia:

Sia data una matrice di N x M numeri interi positivi. Si implementi un algoritmo di backtracking per determinare la lunghezza della più lunga sequenza di numeri adiacenti strettamente crescenti (un numero x è adiacente ad y si trova in alto, in alto a destra, in alto a sinistra, a destra, a sinistra, in basso, in basso a destra, in basso a sinistra rispetto ad y). Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test con il corrispondente output atteso

Soluzione:

Data in input una matrice NxM l'algormito di backtracking esplora tutti i percorsi possibili partendo da ciascuno dei punti della matrice e scendendo in profondità cercando elementi validi nelle otto direzioni. Ogni volta che non trova alcun candidato per proseguire aggiorna il massimo globale se il percorso trovato è più lungo e torna indietro di una mossa proseguendo ad esplorare le direzioni rimanenti delle posizioni precedentemente esplorate. Per ottimizzare la ricerca si può effettuare un pruning dell'albero se si evita di tornare in punti già visitati. Si può quindi mantenere in memoria una seconda matrice NxM inizializzata con tutti zeri in cui salvare a tempo di esecuzione i percorsi parziali più lunghi trovati per ciascun punto. Ogni volta che si processa una soluzione si aggiorna anche la matrice ausiliaria ripercorrendo a ritroso i punti del percorso A individuato e aggiornando i valori presenti nella matrice "memo" se il nuovo sottopercorso è migliore di quelli precedentemente trovati partendo da quella specifica locazione. In questo modo quando si esplora un candidato già visitato basta consultare la matrice ausiliaria e processare il percorso corrente sommato alla sottosoluzione ottima salvata.

Funzioni implementate:

- Il metodo get_candidates(A) ritorna una lista di elementi strettamente maggiori dell'ultimo inserito nel percorso A cercando nelle otto direzioni.
- Il metodo update_memo(A,k) aggiorna la matrice ausiliaria globale ripercorrendo a ritroso i punti del percorso A e aggiornando i valori contenuti in memo con la lunghezza dei nuovi sottopercorsi se sono migliori (sommando k se la soluzione che si sta processando è terminata in un nodo già visitato).
- Il metodo backtracking(A) verifica se l'ultimo elemento inserito in A è già visitato, caso in cui processa la soluzione sommando il sottopercorso. In caso contrario invoca get_candidates() e chiama ricorsivamente backtracking(A) aggiungendo un candidato alla volta.
- Il metodo process_solution(A,k) aggiorna il massimo globale se $length(A) + k$ è maggiore del massimo globale corrente ($k = 0$ se il percorso è terminato per mancanza di candidati, altrimenti k è la lunghezza del sottopercorso che parte dal nodo già visitato in cui è terminata la ricerca).

Complessità temporale:

La Complessità temporale è $O(N^2M^2)$ perché partiamo esplorando in profondità da ogni cella della matrice di input($O(NM)$) e tutte le operazioni fatte all'interno di ogni chiamata ricorsiva, al netto dei vari controlli, sono la get_candidates() che genera al più 8 candidati(tutte le celle adiacenti) e la update_memo() il cui tempo nel caso peggiore potrebbe essere $O(NM)$ perché potrebbe andare a riaggiornare la stessa locazione più volte, in particolare il primo elemento inserito in A, da cui parte la visita in profondità è quello più critico perché basta trovare una nuova soluzione migliorativa anche di 1 per aggiornare in cascata tutti gli elementi presenti nella nuova soluzione. Tuttavia si può intuire che nel momento in cui si giunge ad una nuova soluzione sostanzialmente più lunga rispetto a quella precedentemente trovata, dobbiamo comunque riaggiornare tutti gli elementi contenuti in A, ma questa volta avendo esplorato un numero più grande di

locazioni "nuove" e quindi evitando future visite in quelle locazioni da parte del backtracking, andando a migliorare le performance generali dell'algoritmo pur non potendo garantire un caso peggiore asintoticamente migliore della soluzione brute force.

Problema 2

Traccia:

Sia data un vettore di n numeri interi positivi. Si considerino le sotto-sequenze i cui valori sono ordinati in senso crescente (si assume che non ci siano duplicati). Si scriva un programma per trovare la somma massima tra le somme degli elementi di tali sottosequenza. Ad esempio, se l'array di input è 3, 4, 5, 10, l'output dovrebbe essere 22 (3 + 4 + 5 + 10); se l'array di input è 10, 5, 4, 3, l'output dovrebbe essere 10. Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test con il corrispondente output atteso

Soluzione:

L'approccio forza bruta consiste nel calcolare tutte le sottosequenze ($O(2^n)$) e verificare per ognuna di essa la proprietà di sotto-sequenza crescente e ricercare quella con valore massimo. Tuttavia l'idea per ottimizzare questo tipo di problema consiste nell'individuare la sottostruttura ottima: la sottosequenza del vettore, che parte da indice i, non è altro che il valore arr[i] + la sottosequenza più grande che si può individuare partendo da tutte le locazioni successive ad i. Il problema è stato quindi risolto con un algoritmo di programmazione dinamica che lavora a suffissi. Dato in input un array arr di lunghezza N ogni sottoproblema sul suffisso arr[i:] per i=0...N-1 prova a sommare l'elemento i con la soluzione ottima del sottoproblema su arr[j:] per ogni j=i+1...N-1 se arr[j] maggiore di arr[i] e di questi calcola quindi il massimo. La funzione DP è richiamata partendo da ogni elemento dell'array perché dato che esploriamo solo le sequenze strettamente crescenti non è assicurata la visita esaustiva di tutte le soluzioni, quindi dobbiamo partire con

l'esplorazione da ogni possibile elemento i. L'approccio utilizzato è quello top-down con memoization.

Complessità temporale:

- Numero di sottoproblemi: N, si parte da ogni locazione dell'array a valutare i suffissi.
- Guessing: per ogni sottoproblema arr[i:] si hanno al più N-i scelte possibili (tempo/sottoproblema = $O(N)$)
- Ricorrenza: DP(i) = max(arr[i], DP(j)+arr[i] for j in range(i+1,N) if arr[j] > arr[i]))
- DAG aciclico: tempo totale $O(N^2)$
- Soluzione finale: DP(i) for i in range(N) (extra time: $\Theta(N)$ per valutare il massimo)

Problema 3

Traccia:

Un numero può sempre essere rappresentato come somma di quadrati di altri numeri. Infatti, al limite, il valore 1 è un quadrato e possiamo sempre rappresentare un qualsiasi numero come (1*1 + 1*1 + 1*1 + ...). Dato un numero n, si implementi un algoritmo che trova il numero minimo di quadrati la cui somma è n. Ad esempio, se n = 100, questo valore si può ottenere sommando 52 + 52 + 52 + 52 , quindi usando 4 quadrati, ma anche come 102, quindi usando un solo quadrato. Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test con il corrispondente output atteso

Soluzione:

L'idea è che un numero N può essere scritto come somme di quadrati di numeri al più pari alla propria radice al quadrato, (es. $N = 16 => k = [1...4]$). Il problema può essere risolto con programmazione dinamica riconducendo il problema sull'intero N a più

sottoproblemi identici sull'intero N - k con k = 1...floor(rad(N)). La soluzione ottima del problema originale risulta quindi pari al minimo tra le k scelte ciascuna sommata alla soluzione ottima del sottoproblema N - k. L'approccio utilizzato nell'implementazione è di tipo bottom-up.

Complessità temporale:

- numero sottoproblemi: N
- Guessing: floor(rad(N)) scelte (tempo/sottoproblema: $O(rad(N))$)
- Ricorrenza: DP(i) = min(1+DP(i-k) for k in range(1,int(sqrt(i))+1))
- DAG aciclico: tempo totale $O(N \cdot \sqrt{N})$
- Soluzione finale: DP(N) per l'approccio top-down, DP[N] per l'approccio bottom-up.