Esercizi – Hash Table, Dynamic Programming

Anno Accademico 2022/2023

Dott. Staccone Simone

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Esercizi tabelle di hash

Si consideri una tabella hash di dimensione m = 11 inizialmente vuota. Si mostri il contenuto della tabella dopo aver inserito nell'ordine, i valori 33, 10, 24, 14, 16, 13, 23, 31, 18, 11, 7. Si assuma che le collisioni siano gestite mediante indirizzamento aperto utilizzando come funzione di hash h(k), definita nel modo seguente:

$$h(k) = (h'(k) + 3i + i^2) \bmod m$$
$$h'(k) = k \bmod m$$

Mostrare il contenuto della tabella al termine degli inserimenti e calcolare il numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella.

Esercizi tabelle di hash

Soluzione: La tabella hash e composta nel modo seguente:

0	1	2	3	4	5	6	7	8	9	10
33	23	24	14	11	16	13	18		31	10

Esempio: h(33) = 33 mod 11 + 3*i + i^2 = 0 + 3*i + i^2 = 0 per i = 0

Il valore 7 non può essere inserito, perché la funzione di scan scorre 11 posizioni (molte ripetute) e non trova l'unica cella libera rimasta (la 8). Il numero di accessi medi è 1.2 per i valori che hanno trovato collocazione.

Infatti abbiamo che il numero di accessi (il valore della i + 1) si ricavano per ogni entry della tabella (Esempio: 33 → 1 accesso, infatti i = 0)

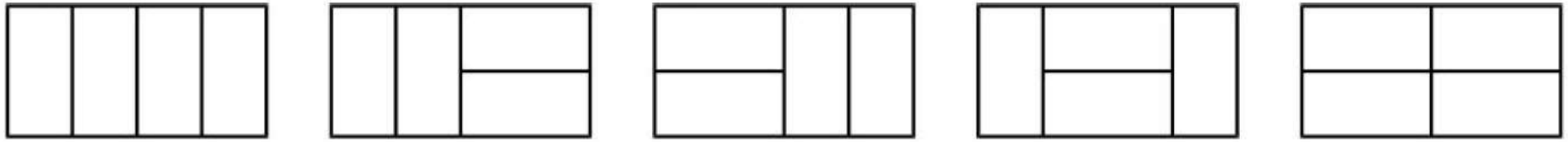
In totale abbiamo : 1 + 1 + 1 + 1 + 2 + 1 + 2 + 1 + 0 + 1 + 1 = 12
Dato che la tabella ha 10 entry occupate il numero medio di accessi è 12/10 = 1,2

Dynamic programming

- Un metodo per spezzare un problema in sottoproblemi in maniera ricorsiva
- Ogni sottoproblema viene risolto una sola volta
- La soluzione di un sottoproblema viene memorizzata in una tabella
- Se si incontra un sottoproblema già risolto si ricava la soluzione dalla tabella
- Si utilizza quindi un approccio ricorsivo e si salva ogni risultato Di una computazione in una tabella, non dovendolo ricomputare nello step successivo della ricorsione.
- La tabella è facilmente indicizzabile (ricerca in O(1))

Dynamic programming - Domino

• Il gioco del domino è basato su tessere di dimensione 2x1. Scrivere un algoritmo che prenda in input un intero *n* e restituisca il numero di possibili disposizioni in un rettangolo 2x*n*
• Fissando *n=4* si ottengono le seguenti soluzioni



• Proviamo a definire una formula ricorsiva per calcolare il numero di disposizioni possibili dato *n:*

- Se non ho tessere, quante sono le disposizioni possibili?
  - una sola disposizione: *nessuna tessera*
- Se ho una tessera, quante sono le disposizioni possibili?
  - una sola disposizione: *una tessera, in verticale*
- Cosa succede se metto l'ultima tessera in verticale?
  - Risolvo il problema di dimensione n − 1
- Cosa succede se metto l'ultima tessera in orizzontale?
  - Ho bisogno di un'altra tessera: risolvo il problema di dimensione n − 2

Dynamic programming - Domino

Scrivendo la ricorrenza in forma algebrica ottengo:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-2) + T(n-1) & n > 1 \end{cases}$$

- Lo riconoscete?

La ricorrenza di Fibonacci

- Qual è la complessità di questo algoritmo quindi?

$O(2^n)$

- Si può fare di meglio?

Si, utilizzando la programmazione dinamica

Dynamic programming - Fibonacci

Algorithm 1: Fibonacci(int n)

$fib[n] \leftarrow 0, ..., 0$
$fib[0] \leftarrow 1$
$fib[1] \leftarrow 1$
for $i \leftarrow 2, i < n, i \leftarrow i + 1$ **do**
  $fib[i] \leftarrow fib[i-2] + fib[i-1]$
end
return fib[n-1]

- Qual è la complessità in *tempo*? T(n) = O(n)
- Qual è la complessità in *spazio*? S(n) = O(n)

Esercizi dynamic programming

Massima sottosequenza crescente

Sia data una sequenza V di n numeri interi distinti. Si scriva una procedura efficiente basata sulla programmazione dinamica per trovare la piu lunga sottosequenza crescente di V (per esempio, se V = [9, 15, 3, 6, 4, 2, 5, 10, 3] allora la piu lunga sottosequenza crescente è: 3, 4, 5, 10).

Esercizi dynamic programming

Denotiamo con DP[i] la dimensione della massima sottosequenza crescente che termina nella posizione i-esima. E` possibile calcolare DP[i] in maniera ricorsiva. Si consideri l'indice i e si considerino gli elementi minori di V [i] nel sottovettore V [1 . . . i − 1]; V [i] può essere utilizzato per estendere la più lunga sottosequenza che termina in uno di questi elementi. Se non esistono elementi minori, allora dobbiamo "ricominciare da capo", ovvero considerare la sequenza composta dal singolo valore V [i]. Un modo per esprimerlo e il seguente:

$$DP[i] = \begin{cases} 1 & i = 1 \\ 1 & i > 1 \textbf{ and } \forall j, 1 \leq j < i : V[i] < V[j] \\ \max\{DP[j] : 1 \leq j \leq i-1 \wedge V[j] < V[i]\} + 1 & \text{altrimenti} \end{cases}$$

Esercizi dynamic programming

Esempio:

V = [1,2,3,2,5,6,7,3,4] → La massima sottosequenza crescente è : [1,2,3,5,6,7]

Applicando la ricorrenza:

DP = [1.2.3.2,4,5,6,3,4]

Esercizi dynamic programming

$$DP[i] = \begin{cases} 1 & i = 1 \\ 1 & i > 1 \textbf{ and } \forall j, 1 \leq j < i : V[i] < V[j] \\ \max\{DP[j] : 1 \leq j \leq i-1 \wedge V[j] < V[i]\} + 1 & \text{altrimenti} \end{cases}$$

Il codice per risolvere questo problema e quindi il seguente, dove il vettore P[i] memorizza l'indice precedente nella sequenza che termina in V[i]. La funzione calcola tutti i valori DP[i] e P[i] con i = 1 . . . n, in ordine. Tutte le volte che viene aggiornato DP[i], viene aggiornato anche P[i]. La variabile max mantiene l'indice del valore in cui termina la piu lunga sottosequenza trovata finora. Al termine del calcolo, viene chiamata la procedura ricorsiva printLongest(), che stampa prima la sottosequenza che termina nel valore di indice P[i] precedente a i, e poi stampa V[i] stesso. Il costo della procedura è $O(n^2)$.

Esercizi dynamic programming

Algorithm 1: maxIncreasingSequance(int V[],int n) : int

```
int DP[n]
int max = 0
for i ← 1 to n do
    DP[i] ← 1
    for j ← 1 to i-1 do
        if (V[j] < V[i]) ∧ (DP[j] + 1 > DP[i]) then
            DP[i] ← DP[j] + 1
        end
    end
    if DP[i] > max then
        max ← DP[i]
    end
end
return max
```

Esercizio 2 – 16/07/2021

Un bambino scende una scala composta da n scalini. Ad ogni passo, può decidere di fare 1, 2, 3 o 4 scalini alla volta. Scrivere un programma in C che determini in quanti modi diversi può scendere le scale. Ad esempio, se n = 7, alcuni dei modi possibili sono i seguenti (rappresentati dalla lunghezza dei passi in numero di scalini):

- 1, 1, 1, 1, 1, 1, 1
- 1, 2, 4
- 4, 2, 1
- 2, 2, 2, 1
- 1, 2, 2, 1

Discutere informalmente la correttezza della soluzione proposta e calcolarne la complessità computazionale.

Esercizio 2 - 16/07/2021

Soluzione:

Il problema può essere risolto utilizzando la programmazione dinamica. Sia T(n) il numero di modi in cui è possibile scegliere n scalini, allora T(n) può essere espresso nel modo seguente:

$$T(n) = \begin{cases} 1 & n = 0 \\ \sum_{k=1}^{\min(n,4)} T(n-k) & n > 0 \end{cases}$$

Esercizio 2 - 16/07/2021

Si può quindi costruire l'algoritmo basato su programmazione dinamica seguente:

Algorithm 1: maxIncreasingSequance(int V[],int n) : int

```
int T[n+1] ← [0, 0,..., 0]
T[1] ← 1
for i ← 2 to n + 1 do
    for k ← 1 to min(i, 4) + 1 do
        T[i] ← T[i] + T[i-k]
    end
end
return T[n]
```

La complessità totale è pari a O(n) il ciclo interno ha un numero di iterazioni massimo limitato a 4 (quindi T(n) = 4n nel caso peggiore).

Approfondimenti

Per capire meglio la programmazione dinamica vi consiglio di vedere anche il problema di Knapsack:
https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Oppure il problema del resto:
https://www.geeksforgeeks.org/understanding-the-coin-change-problem-with-dynamic-programming/

Per il problema del resto esiste anche una soluzione più semplice che si basa su algoritmi greedy, ovvero algoritmi che si 'accontentano' di trovare una soluzione sub-ottima di un problema, ma in alcuni casi (come nel problema del resto) la soluzione sub-ottima coincide con la soluzione ottima

Buona fortuna per l’esame!!!