Esame I.A. 10 luglio 2020

Soluzioni

1 Esercizio 1

Una coda di priorità mi permette di estrarre con costo molto ridotto l'elemento a priorità massima o l'elemento a priorità minima. Tuttavia, nel mio programma, qualche volta mi serve l'elemento a priorità massima e qualche volta quello a priorità minima, potrei dire con uguale probabilità. Per essere il più libero possibile, quindi, voglio individuare una struttura dati che mi consenta di estrarre con costo costante sia l'elemento a priorità massima che quello a priorità minima. Inoltre, vorrei poter pagare un costo basso anche negli inserimenti. Non mi interessa eliminare gli elementi inseriti. Aiutatemi implementando una struttura dati in python che abbia costo di estrazione del massimo e del minimo $O(1)$, e che abbia un costo molto ridotto nell'inserimento.

Soluzione La necessità di restituire in $O(1)$ indifferentemente gli elementi massimo o minimo richiede il mantenimento di riferimenti agli elementi massimo e minimo, che devono essere aggiornati ogni volta che effettuo un nuovo inserimento (non effettuo eliminazioni).

Per poter tenere basso il costo dell'inserimento, scarto qualsiasi struttura dati lineare, poiché il costo dell'inserimento risulterebbe essere $O(n)$, con n il numero degli elementi presenti nella struttura dati.

Adottando una struttura ad albero, posso tentare di mantenere il costo dell'inserimento $O(\log n)$. Un buon inizio potrebbe essere quello di utilizzare un albero binario di ricerca. Posso quindi andare a realizzare una struttura dati basata su un albero binario di ricerca in cui, durante l'inserimento, cerco di capire se sto aggiungendo un nuovo minimo o un nuovo massimo. Il primo caso si verifica nel momento in cui attraverso l'albero andando sempre a sinistra, mentre il secondo caso si verifica nel momento in cui attraverso l'albero andando sempre a destra. Se adotto un approccio iterativo per la funzione di inserimento, posso utilizzare una coppia di flag per tenere traccia di come attraverso l'albero, nella maniera seguente:

```
class MaxMinPrio:
    class Node:
        def __init__(self, val):
            self.val = val
            self.left = None
            self.right = None

    def __init__(self):
        self.root = None
        self.max = None
        self.min = None
```

```
    def getMin(self):
        return self.min

    def getMax(self):
        return self.max

    def insert(self, val):
        leftMost = True
        rightMost = True
        newNode = MaxMinPrio.Node(val)

        x = self.root
        y = None

        while x != None:
            y = x
            if val < x.val:
                rightMost = False
                x = x.left
            else:
                leftMost = False
                x = x.right

        if y is None:
            self.root = newNode
        elif val < y.val:
            rightMost = False
            y.left = newNode
        else:
            leftMost = False
            y.right = newNode

        if leftMost is True:
            self.min = newNode
        if rightMost is True:
            self.max = newNode
```

Chiaramente, l'albero binario di ricerca può degenerare fino ad un costo $O(n)$ per l'inserimento. Potrei quindi prendere in considerazione una struttura dati in grado di ribilanciarsi, come un albero AVL o un albero Rosso-Nero. In entrambi i casi, è sufficiente estendere la funzione di inserimento presentata sopra chiamando le funzioni di fixup viste a lezione a valle dell'inserimento.

2 I Promessi Sposi

Quel ramo del lago di Como, che volge a mezzogiorno, tra due catene non interrotte di monti, tutto a seni e a golfi, a seconda dello sporgere e del rientrare di quelli, vien, quasi a un tratto, a ristringersi, e a prender corso e figura di fiume, tra un promontorio a destra, e un'ampia costiera dall'altra parte; e il ponte, che ivi congiunge le due rive, par che renda ancor più sensibile all'occhio questa trasformazione, e segni il punto in cui il lago cessa, e l'Adda rincomincia, per ripigliar poi nome di lago dove le rive, allontanandosi di nuovo, lascian l'acqua distendersi e rallentarsi in nuovi golfi e in nuovi seni.

Quante volte questo testo contiene la *sottosequenza* "lucia"?

Una stringa w è sottosequenza di un'altra stringa x se w è ottenibile da x cancellando zero o più caratteri di x. Più formalmente, la stringa $w_1 w_2 \dots w_n$ è sottosequenza di un'altra stringa $x_1 x_2 \dots x_m$ se esiste una sequenza strettamente crescente di interi $(k_1, k_2, \dots, k_i)$ con $0 < k_i \leq m$ tale che $w_i = x_{k_i}$.

Scrivere un algoritmo in pseudocodice o in python che prenda in input una stringa testo T di n caratteri e una stringa pattern P di m caratteri e restituisca il numero di volte distinte che la stringa P appare come sottosequenza di T.

Soluzione Dalla definizione di sottosequenza, si potrebbe immediatamente realizzare un algoritmo iterativo che tenta l'eliminazione dei caratteri (o meglio la ricerca delle corrispondenze), ma chiaramente il costo computazionale sarebbe gigantesco. Analizzando meglio il problema, esso può essere scomposto in sottoproblemi enunciandolo come segue:

1. Se gli ultimi due caratteri di T e di P sono uguali abbiamo due possibilità—le dovremo prendere in considerazione entrambe:
   (a) Teniamo traccia di questa uguaglianza ed andiamo a considerare le sottostringhe ottenute da T e da P eliminando l'ultimo carattere di ciascuna.
   (b) Possiamo decidere di ignorare l'ultimo carattere di T e continuare a controllare la sottostringa rimanente.
2. Altrimenti, procediamo a controllare la sottostringa ottenuta ignorando l'ultimo carattere di T.

Il motivo della doppia scelta nel punto 1 si capisce facilmente con un esempio. Se sto cercando "ma" in "$mamma$" e sto partendo dalla fine delle stringhe (così come opera la soluzione identificata poco sopra), non devo prendere il considerazione solo l'occorrenza "$mam\underline{ma}$", ma anche "$\underline{ma}mma$"—e poi tutte le altre.

Un'implementazione in python immediata di questa soluzione è la seguente.

```
def sottostringaRicorsiva(T, P, m, n):
    if (m == 0 and n == 0) or n == 0:
        return 1
    if m == 0:
        return 0
    if T[m - 1] == P[n - 1]:
        return (sottostringaRicorsiva(T, P, m - 1, n - 1) +
                sottostringaRicorsiva(T, P, m - 1, n))
    else:
        return sottostringaRicorsiva(T, P, m - 1, n)
```

Nel primo passo base, consideriamo il caso in cui siamo arrivati alla fine sia di T che di P, oppure alla fine di P. In entrambi i casi, abbiamo trovato un'occorrenza del pattern. Nel secondo passo base, verifichiamo se siamo arrivati alla fine di P, ma non di T, nel qual caso non abbiamo trovato un'occorrenza valida del pattern. La parte finale del codice implementa direttamente l'approccio descritto sopra.

Il problema di questa implementazione è che il costo computazionale è esponenziale. Infatti, è evidente che gli stessi sottoproblemi vengono risolti più volte. Conviene quindi adottare un approccio diverso e cercare di "memorizzare" i risultati dei sottoproblemi già risolti. Viene in aiuto quindi la programmazione dinamica.

Riscriviamo nuovamente il problema, evidenziando la volontà di memorizzare le soluzioni già calcolate in una matrice. Sia $DP[i][j]$ il numero di occorrenze del prefisso j-esimo del pattern $P(j)$ come sottosequenza del prefisso i-esimo del testo $T(i)$. I casi da considerare sono:

- Se il testo è finito ($i = 0$) e il pattern non è vuoto ($j > 0$), non siamo riusciti a trovare il pattern.
- Se il pattern è vuoto ($j = 0$), significa che siamo riusciti a trovare il pattern e lo contiamo come un'occorrenza.
- Se l'ultimo carattere del testo e del pattern sono uguali, possiamo sfruttare quest'uguaglianza oppure no; i due casi vanno sommati (come abbiamo visto in precedenza).
- Se l'ultimo carattere del testo e del pattern sono diversi, ignoriamo l'ultimo carattere del testo.

In maniera più formale:

$$DP[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ e } j > 0 \\ 1 & \text{se } j = 0 \\ DP[i-1][j] + DP[i-1][j-1] & \text{se } i > 0 \text{ e } j > 0 \text{ e } T[i] = P[j] \\ DP[i-1][j] & \text{se } i > 0 \text{ e } j > 0 \text{ e } T[i] \neq P[j] \end{cases}$$

Questa relazione ricorsiva può essere trasformata nel seguente algoritmo—utilizziamo $n+1$ colonne nella matrice DP per assegnare un elemento al concetto di "pattern vuoto" che mal si sposa con l'accesso a carattere, dovendo poi riaggiustare gli indici quando accediamo a P:

Algorithm 1 Promessi Sposi

```
procedure SOTTOSEQUENZA(T, P, m, n)
    DP ← new Matrice m × (n + 1)
    for i ← 1 to m do
        DP[i][0] ← 1
        for j ← 1 to (n + 1) do
            if T[i] == P[j - 1] then
                DP[i][j] ← DP[i - 1][j] + DP[i - 1][j-1]
            else
                DP[i][j] = DP[i-1][j]
    return DP[m][n]
```

Questa soluzione ha costo computazionale $m \cdot n$ e richiede una matrice di dimensione $m \cdot n$, ma si può fare di meglio dal punto di vista spaziale. Infatti, nella trasformazione di questa relazione in un algoritmo, si può effettuare un'ottimizzazione dal punto di vista dello spazio che risulta immediata. Infatti, si può osservare che la riga corrente della matrice DP dipende unicamente dalla riga precedente. Si può quindi sostituire la matrice con una coppia di vettori DP e $DP1$, ottenendo la seguente implementazione in python:

```python
def sottostringa(T, P, m, n):
    DP = [0] * (n + 1)
    DP[0] = 1
    for i in range(0, m):
        DP1 = DP.copy()
        for j in range(1, (n + 1)):
            if T[i] == P[j - 1]:
                DP[j] = DP1[j] + DP1[j - 1]
            else:
                DP[j] = DP1[j]
    return DP[n]
```

È interessante notare come la sottosequenza "lucia" compaia 299837 volte nell'inizio de I Promessi Sposi. Similmente, è interessante analizzare i tempi d'esecuzione delle due implementazioni. Su una

macchina "scadente" (un vecchio Pentium E5400 @ 2.70GHz), l'algoritmo ricorsivo impiega 7.0029 secondi per trovare la soluzione, l'algoritmo basato su DP impiega 0.0012 secondi.

3 Quiz

Data una collezione di elementi, identificati ciascuno da una chiave intera, voglio scegliere una struttura dati appropriata, che minimizzi il costo dell'esecuzione dei miei programmi, prendendo in considerazione l'operazione che svolgo più di frequente. Quale struttura dati sceglierete, nei seguenti casi?

- Inserimento ordinato di un elemento: albero auto bilanciante
- Rappresentazione di un file system: albero n-ario
- Estrazione del valore minimo: heap
- Inserimento in testa: lista singolarmente collegata
- Estrazione dell'elemento k-esimo: array
- Inserimento in coda: lista testa-coda

Dopo l'inserimento in un albero Rosso-Nero:

- ✓ possono essere necessarie fino a 2 rotazioni.
- possono essere necessarie anche più di 2 rotazioni.
- basta al più una rotazione.
- se lo zio è nero, basta ricolorare.