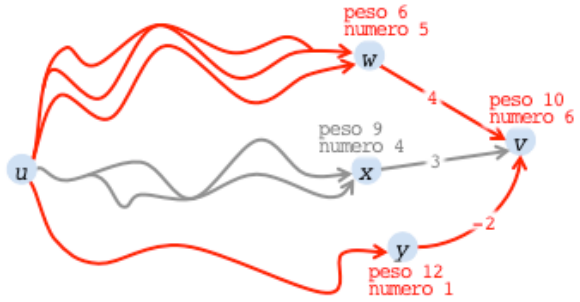
Progettazione di algoritmi

Discussione dell'esercizio [cammini]

Dato un grafo G diretto e pesato con pesi anche negativi, ma senza cicli di peso negativo, e dati due nodi u e v vogliamo contare i cammini minimi da u a v in G. In un qualsiasi cammino minimo da u a v c'è un ultimo arco (w, v) (w potrebbe essere uguale a u). Se sappiamo che ci sono $N[w]$ cammini minimi da u a w, allora ognuno di questi fornisce insieme all'arco (w, v) un cammino minimo da u a v. Per ogni nodo x, sia $M[x]$ il peso di un cammino minimo da u a x. Sia y un qualsiasi nodo per cui esiste l'arco (y, v), se $M[y] + p(y, v) = M[v]$ allora ognuno degli $N[y]$ cammini minimi da u a y induce un (distinto) cammino minimo da u a v.



Per contare i cammini minimi è evidente che dobbiamo conoscere i pesi dei cammini minimi. Se i pesi degli archi fossero positivi, potremmo ordinare i nodi per pesi dei cammini minimi da u non decrescenti e poi contare i cammini minimi dei nodi seguendo un tale ordine, cioè dai nodi più vicini ad u verso quelli più lontani:

$$N[x] = \sum_{y:(y,x)\in E \wedge M[y]+p(y,x)=M[x]} N[y]$$

Ma se ci sono archi di peso negativo, come si vede anche dalla figura sopra, non è garantito che $M[y] + p(y, v) = M[v]$ implichi $M[y] < M[v]$. Possiamo allora seguire l'approccio dell'algoritmo di Bellman-Ford. Consideriamo i sotto-problemi ottenuti limitando il numero di archi dei cammini: per ogni k =0,…,n-1, ed ogni nodo x,

$$N[k,x] = \text{numero cammini da } u \text{ a } x \text{ di peso } M[k,x] \text{ e di lunghezza al più } k$$

Dove M è la stessa tabella dell'algoritmo di Bellman-Ford. Il valore che vogliamo calcolare è $N[n\text{-}1, v]$. I casi base sono $N[0, u] = 1$ e $N[0, x] = 0$ per $x \neq u$. Il calcolo dei rimanenti elementi della tabella N è facile: per ogni $k = 1,\ldots,n\text{-}1$ e ogni nodo x,

$$N[k,x] = \sum_{y:(y,x)\in E \wedge M[k-1,y]+p(y,x)=M[k,x]} N[k-1,y]$$

Il programma che calcola la tabella N, contemporaneamente alla tabella M, è facile da scrivere:

```
CAMMINI(G: grafo diretto e pesato (senza cicli di peso negativo), u,v: nodi)
    M: tabella (n+1)xn
    N: tabella (n+1)xn inizializzata a 0
    FOR ogni nodo x DO M[0, x] <- ∞
    M[0, u] <- 0
    N[0, u] <- 1
    FOR k <- 1 TO n-1 DO
        FOR ogni nodo x DO
            M[k, x] <- M[k-1, x]
            FOR ogni arco (y, x) entrante in x DO
                IF M[k-1, y] + p(y, x) < M[k, x] THEN
                    M[k, x] <- M[k-1, y] + p(y, x)
            FOR ogni arco (y, x) entrante in x DO
                IF M[k-1, y] + p(y, x) = M[k, x] THEN
                    N[k, x] <- N[k, x] + N[k-1, y]
```

```
    RETURN N[n-1, v]
```

La complessità è chiaramente la stessa di quella del calcolo della sola tabella *M* (almeno asintoticamente) e quindi è O(*nm*). Non è necesssario mantenere le intere tabelle ma solmente le ultime due righe di ognuna, perciò è sufficiente memoria O(*n*), anziché O(n^2).

Cammini minimi tra tutte le coppie di nodi

Se vogliamo calcolare i cammini minimi tra tutte le coppie di nodi, gli algoritmi che conosciamo risultano piuttosto inefficienti. Se il grafo ha solamente pesi non negativi potremmo usare l'algoritmo di Dijkstra che applicandolo a partire da ogni nodo richiede tempo O($n(n + m)\log n$). Se però il grafo ha anche pesi negativi, possiamo usare l'algoritmo di Bellman-Ford che applicato a partire da ogni nodo richiede O(n^2m). Se il grafo è denso, cioè ha $\Omega(n^2)$ archi, la complessità diventa O(n^4). Tuttavia c'è un algoritmo più efficiente che calcola i cammini minimi tra tutte le coppie di nodi ed è anch'esso basato sulla programmazione dinamica.

Come al solito, l'algoritmo si basa sulla considerazione di certi sotto-problemi. Nel caso dell'algoritmo di Bellman-Ford è stato limitato il numero di archi in questo caso limitiamo invece i nodi intermedi. Assumiamo che i nodi siano numerati da 1 a *n*. Inoltre, almeno per ora, assumiamo che il grafo non contenga cicli di peso negativo, per cui sappiamo che i cammini minimi esistono e sono semplici. Per ogni coppia di nodi *i* e *j* (anche uguali) e ogni $k = 0,1, \ldots,n$, definiamo:

$$F[i,j,k] = \begin{cases} \text{minimo peso di un cammino da } i \text{ a } j \text{ con nodi intermedi in } \{1,\ldots,k\} & \text{se c'è un cammino da } i \text{ a } j \\ & \text{che passa per nodi} \\ & \text{intermedi in } \{1,\ldots,k\} \\ \infty & \text{altrimenti} \end{cases}$$

Chiaramente *F*[*i*, *j*, *n*] è il peso minimo di un cammino da *i* a *j*. Per il caso base, cioè $k = 0$, intendiamo che i cammini non possono passare per nodi intermedi, quindi

$$F[i,j,0] = \begin{cases} p(i,j) & \text{se } i \neq j \text{ e } (i,j) \in E \\ 0 & \text{se } i = j \\ \infty & \text{altrimenti} \end{cases}$$

Vediamo ora come possiamo calcolare *F*[*i*, *j*, *k*+1] conoscendo i valori *F*[·, ·, *k*]. Un cammino da *i* a *j* che può toccare come nodi intermedi solamente i nodi in {1,…,*k*+1}, può essere di due tipi: o non passa per il nodo *k*+1 o passa per il nodo *k*+1. Se è del secondo tipo, cioè passa per il nodo *k*+1, sarà costituito da un cammino C_1 che va da *i* a *k*+1 e da un cammino C_2 che va da *k*+1 a *j*. Siccome i cammini sono semplici sia C_1 che C_2 non possono toccare come nodo intermedio il nodo *k*+1. Quindi il peso minimo dei cammini del primo tipo è dato da *F*[*i*, *j*, *k*] mentre quello dei cammini del secondo tipo è dato da *F*[*i*, *k*+1, *k*] + *F*[*k*+1, *j*, *k*]. Perciò per ogni $k = 0,\ldots n$-1, abbiamo che

$$F[i,j,k+1] = \min\{F[i,j,k],\ F[i,k+1,k] + F[k+1,j,k]\}$$

Quello che abbiamo appena delineato è noto come algoritmo di *Floyd-Warshall*. Il programma che calcola la tabella è facile da scrivere:

```
FLOYD_WARSHALL(G: grafo diretto e pesato)
    F: tabella nxnx(n+1) inizializzata a ∞
    FOR i <- 1 TO n DO F[i, i, 0] <- 0
    FOR ogni arco (i, j) DO F[i, j, 0] <- p(i, j)
    FOR k <- 1 TO n DO
        FOR i <- 1 TO n DO
            FOR j <- 1 TO n DO
                F[i, j, k] <- F[i, j, k-1]
                IF F[i, k, k-1] + F[k, j, k-1] < F[i, j, k] THEN
                    F[i, j, k] <- F[i, k, k-1] + F[k, j, k-1]
    RETURN F[·, ·, n]
```

Chiaramente la complessità dell'algoritmo è O(n^3). La memoria usata è anch'essa O(n^3), però possiamo risparmiarla. Infatti, tenendo conto che $F[i, k, k\text{-}1] = F[i, k, k]$ e $F[k, j, k\text{-}1] = F[k, j, k]$, l'algoritmo può essere riscritto in modo equivalente usando solamente una tabella di dimensione O(n^2):

```
FLOYD_WARSHALL(G: grafo diretto e pesato)
    F: tabella nxn inizializzata a ∞
    FOR i <- 1 TO n DO F[i, i] <- 0
    FOR ogni arco (i, j) DO F[i, j] <- p(i, j)
    FOR k <- 1 TO n DO
        FOR i <- 1 TO n DO
            FOR j <- 1 TO n DO
                IF F[i, k] + F[k, j] < F[i, j] THEN
                    F[i, j] <- F[i, k] + F[k, j]
    RETURN F
```

Tramite una tabella aggiuntiva che memorizza per ogni coppia i, j l'indice k dell'ultimo aggiornamento di $F[i, j]$ (che corrisponde all'indice più alto di un nodo intermedio nel cammino minimo da i a j), è possibile trovare anche i cammini minimi oltre ai loro pesi. Ma non vedremo i dettagli di ciò.

Se il grafo di input contiene cicli di peso negativo, l'algoritmo di Floyd-Warshall permette di rilevarli. Infatti, se non ci sono, per ogni i, avremo che $F[i, i] = 0$. Mentre se c'è un ciclo di peso negativo allora per ogni nodo i che appartiene a un tale ciclo risulterà $F[i, i] < 0$, perchè c'è un cammino da i a i di peso negativo.

Problemi relativi a sequenze

Ci sono molti problemi che coinvolgono sequenze o array e che possono essere efficientemente risolti con la Programmazione Dinamica. Per la maggior parte di questi, i sottoproblemi che risultano convenienti per l'applicazione della Programmazione Dinamica sono relativi a sottoarray o sottosequenze di elementi consecutivi che spesso sono semplicemente i prefissi delle sequenze o array originali. Inizieremo con una nostra vecchia conoscenza.

Il problema del massimo sottovettore: l'algoritmo ottimale

Ricordiamo che il problema del massimo sottovettore consiste nel trovare, dato un vettore A di n numeri reali, un sottovettore (una sequenza di elementi consecutivi del vettore) la cui somma degli elementi è massima. Si assume che il sottovettore vuoto abbia somma zero. Abbiamo visto un algoritmo basato sulla tecnica Divide et Impera che risolve il problema in O($n$log$n$). Ora vogliamo applicare la tecnica della Programmazione Dinamica. La scelta più semplice e naturale dei sotto-problemi è quella di considerare i sottovettori prefisso del vettore A, cioè i sottovettori A[1⋯i], per i = 1,…,n. In relazione a questi sotto-problemi dobbiamo calcolare la seguente tabella:

$$M[i] = \text{massima somma di un sottovettore di } A[1 \cdots i]$$

Il caso base è facile: $M[1] = \max\{A[1], 0\}$. Vediamo ora come calcolare $M[i]$ sfruttando la conoscenza di $M[j]$ per $j < i$. Il massimo sottovettore di $A[1…i]$ può essere di due tipi o non contiene l'elemento $A[i]$ o lo contiene. Per il primo tipo il valore massimo è dato da $M[i - 1]$. Ma per il secondo tipo come calcoliamo il massimo? Tale massimo è uguale a $A[i] + S$, dove S è la massima somma di un sottovettore che termina nell'elemento $A[i - 1]$ (o e vuoto). La domanda a questo punto è come calcoliamo S? È chiaro che potremmo farlo direttamente, ma questo richiederebbe tempo O(i) e quindi l'intero algoritmo richiederebbe O(n^2). Allora, l'unica speranza è di calcolarlo in modo più efficiente sfruttando i valori $M[j]$ per $j < i$. Ma non si riesce a vedere nessun modo di farlo. Dobbiamo quindi cambiare i sotto-problemi e la tabella M. Quello che invero ci occorre è conoscere il massimo relativamente ai sottovettori che terminano in una posizione i:

$$M[i] = \text{massima somma di un sottovettore di } A \text{ che termina in } i$$

Il caso base è $M[1] = A[1]$. Vediamo ora come calcolare $M[i]$ per $i > 1$:

$$M[i] = \max\{A[i],\ A[i] + M[i - 1]\}$$

Il valore massimo tra tutti i sottovettori di A è semplicemente:

$$\max\{\max_{i=1,\ldots,n} M[i], \quad 0\}$$

Il programma che calcola la tabella e il valore massimo è immediato:

```
MS(A: array di n numeri)
    M: tabella di dimensione n
    M[1] <- A[1]
    max <- max{M[1], 0}
    FOR i <- 2 TO n DO
        M[i] <- max{A[i], A[i] + M[i-1]}
        IF M[i] > max THEN max <- M[i]
    RETURN max
```

La complessità dell'algoritmo è O(*n*) quindi è ottimale. Osserviamo che possiamo evitare di mantenere la tabella ci basta l'ultimo valore calcolato:

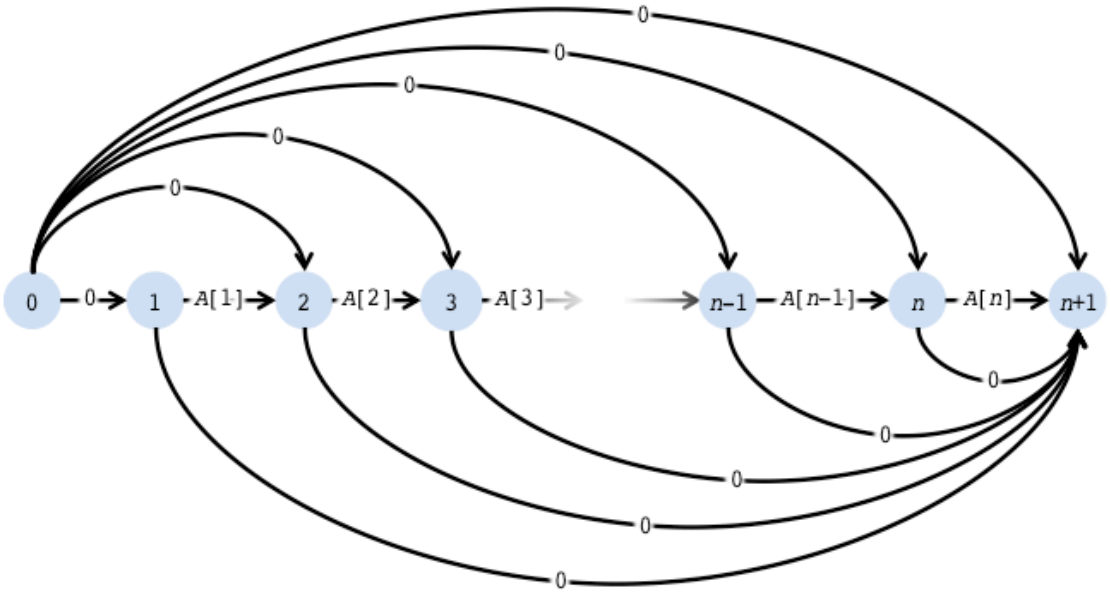
```
MS(A: array di n numeri)
    m <- A[1]
    max <- max{m, 0}
    FOR i <- 2 TO n DO
        m <- max{A[i], A[i] + m}
        IF m > max THEN max <- m
    RETURN max
```

Per ricostruire anche il sottovettore di somma massima è sufficiente registrare l'ultimo indice *i* che ha portato ad aggiornare il valore di `max` . Tale indice ci dà l'estremo destro del sottovettore massimo, l'estremo sinistro può essere calcolato direttamente:

```
MS_SOL(A: array di n numeri)
    m <- 0
    b <- 0      /* Inizializzazione dell'estremo destro del sottovettore */
    max <- 0
    FOR i <- 1 TO n DO
        m <- max{A[i], A[i] + m}
        IF m > max THEN
            max <- m
            b <- i
    IF b = 0 THEN RETURN 0, 0, 0    /* Sottovettore vuoto */
    m <- A[b]
    a <- b              /* Inizializzazione dell'estremo sinistro del sottovettore */
    WHILE m < max DO    /* Calcola il massimo sottovettore che termina in b */
        a <- a - 1
        m <- m + A[a]
    RETURN a, b, max
```

`MS_SOL(A)` ritorna gli estremi di un sottovettore di somma massima del vettore *A* e il valore della somma. Se il massimo sottovettore è vuoto, entrambi gli estremi ritornati sono 0. La complessità temporale è O(*n*) e la memoria usata è O(1), quindi l'algoritmo è ottimale sia in tempo che in spazio di memoria.

C'è un modo alternativo di trovare un algoritmo di Programmazione dinamica per il problema del massimo sottovettore basato sulla rappresentazione tramite un opportuno grafo. Possiamo introdurre uno stato iniziale che denotiamo con un nodo 0 da cui è possibile decidere la posizione d'inizio del sottovettore tramite archi di peso 0. Per ogni posizione *i* c'è un nodo corrispondente e un arco che va alla posizione successiva di peso *A*[*i*]. Infine un nodo finale *n*+1 a cui da ogni posizione è possibile andare con peso 0:



Nel grafo G_A così definito, ogni cammino dal nodo 0 al nodo n+1 corrisponde alla scelta di un sottovettore di A, compreso il sottovettore vuoto. Il peso del cammino è proprio la somma del sottovettore. Quindi il problema diventa quello di calcolare un cammino di peso massimo (longest path) dal nodo 0 al nodo n+1 del grafo G_A. Siccome il grafo è un DAG ed ha O(n) archi, otteniamo un algoritmo di complessità ottimale O(n). Si osservi inoltre che se $L[i]$ è il massimo peso di un cammino dal nodo 0 al nodo i questo equivale a max{$M[i]$, 0}. Quindi ritroviamo essenzialmente gli stessi sotto-problemi che abbiamo già trovato per altre vie.

Il problema della massima sottosequenza comune (LCS)

La maggior parte degli strumenti per confrontare file, come ad esempio l'utility *diff*, sono basati sul calcolo della massima sottosequenza comune dei file. Nell'esempio qui sotto i caratteri della massima sottosequenza comune delle due linee di testo sono stati sottolineati:

```
La massima sotto-sequenza in comune.

Il problema della Massima Sottosequenza Comune.
```

La formulazione del problema è la seguente:

> Date due sequenze $X = (x_1,...,x_n)$ e $Y = (y_1,...,y_m)$, trovare una sottosequenza comune di lunghezza massima.

In inglese è conosciuto con il nome di *Longest Common Subsequence* (*LCS*) problem. Una sottosequenza comune (di lunghezza k) è data da una sequenza di indici di X $1 \le i_1 < i_2 < \cdots < i_k \le n$ e una sequenza di indici di Y $1 \le j_1 < j_2 < \cdots < j_k \le m$, tali che

$$x_{i_1} = y_{j_1},\ x_{i_2} = y_{j_2}, \ldots,\ x_{i_k} = y_{j_k}$$

È chiaro che una ricerca esaustiva prende tempo esponenziale in n e m. Infatti tutte le possibili sottosequenze di X sono esattamente 2^n (se consideriamo anche la sottosequenza vuota) perchè i sottoinsiemi di {1,2,…,n} corrispondono biunivocamente alle sottosequenze di X (ogni sottoinsieme è la sequenza degli indici di una sottosequenza). La stessa cosa vale ovviamente per Y che ha quindi 2^m sottosequenze.

Come al solito preoccupiamoci di calcolare solamente la massima lunghezza di una sottosequenza comune e vediamo poi come ricostruire una tale sottosequenza. Volendo applicare la Programmazione Dinamica, dobbiamo

individuare i sotto-problemi da considerare. Una possibilità che viene subito in mente sono i prefissi delle due sequenze, cioè consideriamo i sotto-problemi relativi ai prefissi $X_i = (x_1,\ldots,x_i)$ e $Y_j = (y_1,\ldots,y_j)$ per $i = 0,\ldots,n$ e $j = 0,\ldots,m$:

$$L[i,j] = \text{massima lunghezza di una sottosequenza comune a } X_i \text{ e } Y_j$$

dove X_0 e Y_0 denotano i prefissi vuoti. Chiaramente il valore che ci interessa è $L[n, m]$. I casi base si hanno quando $i = 0$ o $j = 0$: per $i = 0,\ldots,n$ e $j = 0,\ldots,m$,

$$L[0,j] = 0 \quad \text{e} \quad L[i,0] = 0$$

Vediamo come calcolare $L[i, j]$ quando sia i che j sono almeno 1. Cerchiamo di ricondurre tale calcolo a valori di L per prefissi più piccoli. Una massima sottosequenza comune a X_i e Y_j come può essere fatta? Se non comprende l'ultimo elmento, x_i, di X_i, $L[i, j]$ è uguale a $L[i\text{-}1, j]$. Simmetricamente, se non comprende y_j, $L[i, j]$ è uguale a $L[i, j\text{-}1]$. Altrimenti comprende sia x_i che y_j. Ma questo è possibile solo se $x_i = y_j$. In quest'ultimo caso la sottosequenza è formata da una sottosequenza comune a X_{i-1} e Y_{j-1} e dall'elemento $x_i = y_j$. Quindi $L[i, j]$ è uguale a $L[i\text{-}1, j\text{-}1] + 1$. Ricapitolando abbiamo determinato il seguente metodo di calcolo: per $i = 1,\ldots,n$ e $j =1,\ldots,m$

$$L[i,j] = \begin{cases} \max\{L[i-1,j],\ L[i,j-1],\ L[i-1,j-1]+1\} & \text{se } x_i = y_j \\ \max\{L[i-1,j],\ L[i,j-1]\} & \text{altrimenti} \end{cases}$$

È facile vedere che se $x_i = y_j$, allora

$$L[i-1,j] \leqslant L[i-1,j-1]+1 \quad \text{e} \quad L[i,j-1] \leqslant L[i-1,j-1]+1$$

Quindi il calcolo di $L[i, j]$ può essere leggermente semplificato:

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & \text{se } x_i = y_j \\ \max\{L[i-1,j],\ L[i,j-1]\} & \text{altrimenti} \end{cases}$$

Il programma che calcola la tabella procede dai prefissi più corti verso quello più lunghi:

```
LCS(X: sequenza lunga n, Y: sequenza lunga m)
    L: tabella (n+1)x(m+1)
    FOR i <- 0 TO n DO L[i, 0] <- 0
    FOR j <- 0 TO m DO L[0, j] <- 0
    FOR i <- 1 TO n DO
        FOR j <- 1 TO m DO
             IF X[i] = Y[j] THEN
                 L[i, j] <- L[i-1, j-1] + 1
             ELSE
                 L[i, j] <- max{L[i-1, j], L[i, j-1]}
    RETURN L[n, m]
```

La complessità dell'algoritmo è O(nm). Se siamo interessati solamente alla lunghezza della massima sottosequenza, è sufficiente mantenere solamente le ultime due righe della tabella L, quindi possiamo usare memoria O(m), anziché O(nm). Se invece si vuole ricostruire la sottosequenza comune, come al solito, è sufficiente ripercorrere a ritroso le scelte che hanno portato a calcolare la lunghezza massima.

```
LCS_SOL(X: sequenza lunga n, Y: sequenza lunga m, L: tabella)
    SOL <- lista vuota
    i <- n
    j <- m
    WHILE i > 0 AND j > 0 DO
        IF X[i] = Y[j] THEN
            SOL <- (i, j, X[i]) + SOL   /* Aggiunge in testa alla lista SOL */
            i <- i - 1
            j <- j - 1
```

```
        ELSE IF L[i, j] = L[i - 1, j] THEN
            i <- i - 1
        ELSE
            j <- j - 1
    RETURN SOL
```

La lista ritornata contiene per ogni elemento della sottosequenza l'indice di X, l'indice di Y e l'elemento. La complessità è O($n + m$) perchè ad ogni iterazione del `WHILE` uno almeno dei due indici `i , j` è decrementato.

Esercizio [prezzi]

Dato un vettore P di n interi positivi in cui $P[t]$ rappresenta il prezzo di una certa merce nel giorno t, si vuole sapere qual'è il giorno i in cui conviene comprare la merce ed il giorno j, con $j > i$, in cui conviene rivenderla in modo da massimizzare il profitto o in alternativa (se non è possibile ottenere un guadagno) minimizzare la perdita. In altre parole siamo interessati a conoscere la coppia di giorni (i, j) con $i < j$ per cui risulta massimo il valore $P[j] - P[i]$. Descrivere un algoritmo che risolve il problema in tempo O(n).