Dispensa di Esercizi di Algoritmica (Parte II)

Esercizi su hashing, alberi, grafi, programmazione dinamica e complessità (v1.0)

Chiara Baraglia, Andrea Lisi, Nicolas Manini, Davide Rucci

29 maggio 2019

Sommario

Questa dispensa contiene esercizi svolti utili alla preparazione dell'esame di Algoritmica. In particolare vi sono esercizi sulla seconda parte del corso che comprende l'ordinamento in tempo lineare, le strutture dati come dizionari, alberi, grafi, programmazione dinamica e complessità. Ogni esercizio è classificato per argomento in modo da guidare il lettore verso gli argomenti su cui vuole esercitarsi. Tutte le soluzioni sono dettagliatamente commentate per facilitarne la comprensione. Gli esercizi sono stati tratti principalmente da temi d'esame o compitini passati. Si prega di segnalare eventuali errori a n.manini@studenti.unipi.it, d.rucci1@studenti.unipi.it o andrealisi.12lj@gmail.com.

Esercizio 1 (Compitino 30/05/2018). `Programmazione Dinamica`

Si definisca la relazione che consente di calcolare mediante Programmazione Dinamica la *Longest Common Subsequence* di due stringhe S_1 e S_2, e la si applichi alle due stringhe $S_1 = BDEH$, $S_2 = BCDH$.

Soluzione.

Il problema di Longest Common Subsequence (LCS) consiste di trovare la pià lunga sottosequenza comune in 2 (o più) sequenze di input. L'ordine relativo deve essere rispettato, ma le sotto-sequenze non devono per forza essere composte da caratteri contigui. Per esempio *ace* e *abc* sono entrambe sotto-sequenze di *abcdefg*, ma *acb* no.

La regola ricorsiva che consente di calcolare la LCS tra due sequenze è:

$$M[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0 \\ M[i-1, j-1] + 1 & \text{se } i > 0,\ j > 0,\ S_1[i] = S_2[j] \\ max\{M[i-1,j], M[i,j-1]\} & \text{altrimenti} \end{cases}$$

La tabella di programmazione dinamica che si ottiene applicando questa regola ricorsiva sulle due stringhe fornite è dunque:

		0	1	2	3	4
			B	D	E	H
0		0	0	0	0	0
1	B	0	1	1	1	1
2	C	0	1	1	1	1
3	D	0	1	2	2	2
4	H	0	1	2	2	**3**

La lunghezza della LCS è $M[4,4] = 3$. La complessità in tempo di questo algoritmo è $T(n,m) = \Theta(nm)$. □

Esercizio 2 (Compitino 30/05/2018 - II). `Hashing`
Sia dato l'insieme di chiavi $S = \{2, 4, 5, 13, 14, 16\}$. Inserirle in una tabella hash di dimensione $m = 11$ in cui le collisioni sono gestite con indirizzamento aperto e hash doppio.

Soluzione.
Forniamo come soluzione una tabella che contiene le sequenze di scansione utilizzate per inserire ogni chiave. Poiché non specificate, stabiliamo prima di tutto quali funzioni hash utilizzare.

$$h_1(k) = k \mod m = k \mod 11$$
$$h_2(k) = 1 + (k \mod (m-1)) = 1 + (k \mod 10)$$
$$h(k,i) = [h_1(k) + ih_2(k)] \mod m$$

Si noti che la h_1 è stata scelta tra le funzioni hash base, mentre h_2 è definita appositamente in modo da non avere mai 0 o un numero maggiore di 11 come risultato. In particolare è importante che il risultato di h_2 non sia nullo per evitare di dover fare passi di lunghezza zero nella sequenza di scansione di una chiave.

Chiave K	$\mathbf{h_1}$	$\mathbf{h_2}$	**Sequenza di Scansione**
2	2	–	2
4	4	–	4
5	5	–	5
13	2	4	2, 6
14	3	–	3
16	5	7	5, 1

Per cui la tabella finale sarà:

	16	**2**	**14**	**4**	**5**	**1**	**3**			
0	1	2	3	4	5	6	7	8	9	10

□

Esercizio 3 (Compitino 30/05/2018 - III). 🏷 `Alberi`

Sia dato un albero binario con interi memorizzati nei nodi. Si progetti un algoritmo ricorsivo che restituisce come risultato il conteggio del numero di foglie la cui chiave è il doppio di quella del padre.

Soluzione.

Questa soluzione assume di avere a disposizione il campo `parent` in ogni nodo dell'albero.

```
DoppioValore(u)
    if u = NIL
        return 0
    if u.left = NIL && u.right = NIL // Se il nodo u e' una foglia
        if u.parent ≠ NIL && u.parent.key = 2*u.key
            // La radice non ha un padre (si verifica se l'albero contiene un solo nodo)
            return 1
        else
            return 0

    return DoppioValore(u.left) + DoppioValore(u.right)
```

La complessità di questa soluzione è lineare nel numero di nodi nell'albero, in quanto ognuno di essi viene esaminato in tempo costante e una volta soltanto. Dunque $T(n) = \Theta(n)$.
Si noti che era possibile risolvere questo problema, mantenendo la stessa complessità, anche senza utilizzare i puntatori al padre, facendo invece uso del passaggio di parametri durante la ricorsione:

```
DoppioValore(u, parent) // Prima chiamata: DoppioValore(T.root, NIL)
    if u = NIL
        return 0
    if u.left = NIL && u.right = NIL // Se il nodo u e' una foglia
        if parent ≠ NIL && parent.key = 2*u.key
            // La radice non ha un padre (si verifica se l'albero contiene un solo nodo)
            return 1
        else
            return 0

    return DoppioValore(u.left, u) + DoppioValore(u.right, u) // Il nodo corrente diventa parent
        dei figli
```

□

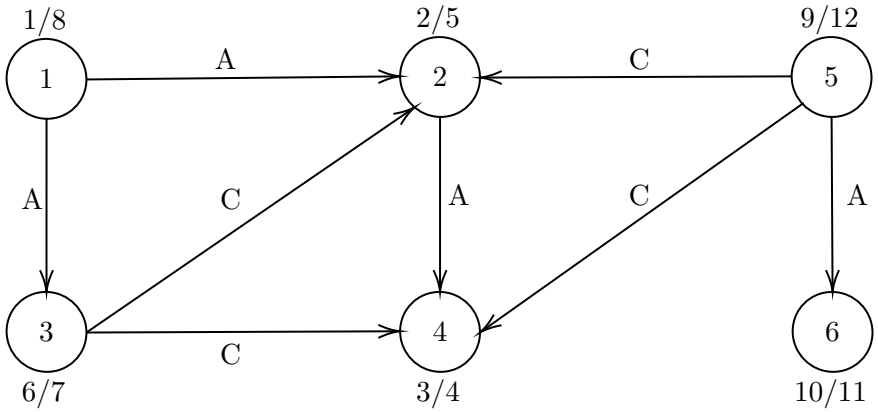
Esercizio 4 (Compitino 30/05/2018 - IV). 🏷 `Grafi`

Sia dato un grafo $G = (V, E)$ orientato con 6 vertici e i seguenti archi:

$$E = \{(1,2), (1,3), (2,4), (3,2), (3,4), (5,2), (5,4), (5,6)\}.$$

Si assuma che le liste di adiacenza siano ordinate in modo crescente per vertice destinazione di ogni arco. Calcolare la visita *DFS* su G e mostrare l'albero DFS specificando anche il tipo di archi individuati dalla visita (ossia dell'albero, indietro, in avanti e attraversamento).

Soluzione.
Per questo tipo di esercizi è sufficiente disegnare il grafo con associate le varie marche temporali e i tipi di arco trovati.



Dove A denota un arco di albero e C un arco di attraversamento (crossing).

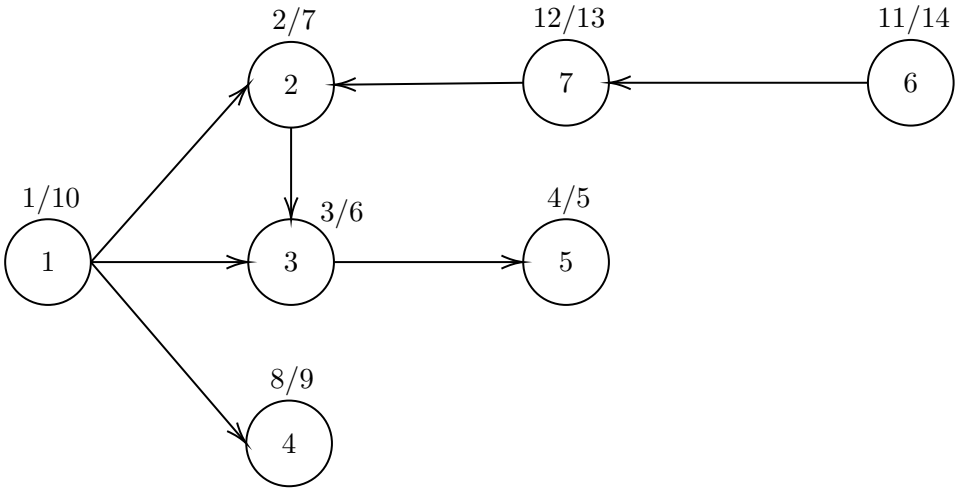
□

Esercizio 5 (Esame 22/06/2018). `Grafi`
Sia dato il grafo diretto aciclico $G(V, E)$ costituito da 7 vertici numerati da 1 a 7 e i seguenti archi:

$$E = \{(1,2), (1,3), (1,4), (2,3), (4,3), (3,5), (6,7), (7,2)\}.$$

Si calcoli l'ordinamento topologico di G simulando il funzionamento dell'algoritmo TOPOLOGICAL_SORT, assumendo che le liste di adiacenza siano ordinate per vertice destinazione.

Soluzione.
Per prima cosa dobbiamo effettuare una DFS sul grafo, assegnando i seguenti valori temporali ai nodi (apertura/chiusura della chiamata ricorsiva).



Una volta effettuata la visita basta ordinare i vertici per tempo finale decrescente, ottenendo quindi l'ordinamento dei nodi $6, 7, 1, 4, 2, 3, 5$. □

Esercizio 6 (Esame 13/07/2018). 🏷 `Grafi`

Si modifichi l'algoritmo BFS in modo tale *BFS(G)* visiti tutti i nodi del grafo formando una foresta di alberi BF.

Soluzione.

```
BFS_Forest(G)
    for all v ∈ V
        v.color = WHITE
        v.π = NIL
        v.d = ∞

    for all v ∈ V
        if v.color = WHITE
            BFS(G, v)
```

Inoltre è necessario rimuovere il ciclo iniziale di colorazione dei nodi dalla procedura $BFS(G, v)$ vista a lezione in quanto i nodi devono essere colorati una sola volta e non ogni volta che la BFS viene eseguita. In questo modo la procedura BFS_Forest può accorgersi di quali siano i nodi non ancora esplorati dopo ogni BFS. La complessità di questa soluzione è la stessa della BFS standard, in quanto viene aggiunto all'inizio solo un ciclo sui vertici. Si noti che, per quanto la chiamata a $BFS(G, v)$ sia annidata all'interno di un ciclo su tutti i vertici, questa esplori comunque una sola volta ciascun nodo del grafo (cioè i nodi già visitati in precedenza non vengono visitati ancora da una successiva chiamata). Per cui $T(|V|, |E|) = \Theta(|V| + |E|)$. □

Esercizio 7 (Esame 13/07/2018 - II). 🏷 `Complessità`

1. Si dia la complessità in tempo al caso pessimo dell'algoritmo di programmazione dinamica per la soluzione del problema dello Zaino, e si motivi la risposta.
2. La complessità del punto precedente è di tipo polinomiale? (Si motivi la risposta.)

Soluzione — Punto 1.

La complessità di questo algoritmo è $T(n, W) = \Theta(nW)$, dove n è il numero totale di oggetti disponibili e W la capacità complessiva dello zaino. Questa complessità è dovuta alla costruzione della matrice di programmazione dinamica che, per questo problema, è formata da $n + 1$ righe e $W + 1$ colonne. La matrice deve essere riempita a partire dalla posizione $[0, 0]$ secondo la seguente regola ricorsiva:

$$M[i, j] = \begin{cases} 0, & \text{se } i = 0 \text{ o } j = 0 \\ M[i-1, j] & \text{se } w_i > j \\ \max\{M[i-1, j], M[i-1, j-w_i] + v_i\} & \text{se } w_i \leq j \end{cases}$$

dove w_i è il peso dell'i-esimo oggetto, e v_i il suo valore. □

Soluzione — Punto 2.

La complessità del punto precedente è tipo *pseudopolinomiale*, ovvero è polinomiale nel *valore* di W,

ma non nella dimensione dell'input. Infatti il valore di W è esponenziale nella sua rappresentazione. Ad esempio, preso $W = 512$, dobbiamo riempire una matrice di 2^9 colonne (e non 9, che sarebbe la lunghezza della rappresentazione di 512 in binario), mentre il numero n di oggetti rappresenta realmente una "dimensione" dell'input. □

Esercizio 8 (Esame 27/06/2016). 🏷 `Alberi`

Si definisce *larghezza* di un albero binario il numero massimo di nodi che stanno tutti sul medesimo livello. Progettare e descrivere in pseudocodice un algoritmo efficiente che calcoli la larghezza di un albero binario di n nodi. Analizzare la complessità in tempo e in spazio dell'algoritmo proposto.

Soluzione.

La soluzione più semplice è quella di visitare l'albero per livelli e tenere traccia del numero di nodi incontrati ad ogni livello. Se dovesse essere maggiore del massimo incontrato finora, questo sarà il nuovo valore della larghezza dell'albero.

```
1  Larghezza(T)
2      if T = ∅ // Controllo se l'albero e' vuoto
3          return 0
4
5      larghezza = 1 // Massima larghezza dei livelli visitati
6      livello = 0 // Livello correntemente analizzato, la radice ha livello 0
7      ctr = 1 // Sul livello 0 c'e' un nodo (la radice non e' NIL)
8
9      u = T.root
10     u.d = 0
11     Q = [] // Istanzio una nuova coda vuota
12     Enqueue(Q, u)
13     while Q ≠ ∅
14         v = Dequeue(Q)
15         if v.d ≠ livello // Sono sceso di un livello
16             livello = v.d // Aggiorno il livello corrente
17             ctr = 0
18
19         ctr = ctr + 1
20         if ctr > larghezza // Ho trovato un nuovo livello piu' popolato
21             larghezza = ctr
22
23         // Accodo l'analisi dei figli
24         if v.left ≠ ∅ // Se v ha un figlio sinistro
25             v.left.d = v.d + 1 // Il figlio ha profondita' maggiore
26             Enqueue(Q, v.left)
27         if v.right ≠ ∅ // Se v ha un figlio destro
28             v.right.d = v.d + 1 // Il figlio ha profondita' maggiore
29             Enqueue(Q, v.right)
30
31     // Ho esaminato tutti i livelli
32     return larghezza
```

La complessità in tempo di questo algoritmo è $T(n) = \Theta(n)$ perché ogni nodo viene esaminato una sola volta durante la visita per livelli, mentre quella in spazio è $S(n) = \Theta(n)$ ed è dovuta sia all'utilizzo della coda, sia alla memorizzazione del campo d in ogni nodo dell'albero (è considerato spazio aggiuntivo in quanto è un valore utilizzato dall'algoritmo e non già presente nell'albero al momento dell'input). □

Esercizio 9 (Esame 27/06/2016 - II). 🏷 `Complessità` 🏷 `Relazioni di Ricorrenza`
Impostare e risolvere la ricorrenza che descrive il costo in tempo dell'algoritmo `GeneraBinarie`:

```
1 GeneraBinarie(B, k)
2     if k = n
3         Stampa(B)
4     else
5         B[k+1] = 0
6         GeneraBinarie(B, k+1)
7         B[k+1] = 1
8         GeneraBinarie(B, k+1)
```

Soluzione.
La ricorrenza è definita, per una costante c, come:

$$T(i) = \begin{cases} Costo(Stampa(B)), & \text{se } i = n \\ 2T(i+1) + c, & \text{se } 0 \leq i < n \end{cases}$$

Procediamo a risolvere la ricorrenza in modo iterativo:

$$\begin{aligned} T(0) &= 2T(1) + c = 2[2T(2) + c] + c = \\ &= 2^2 T(2) + 3c = 8T(3) + 7c = \\ &= \cdots = 2^i T(i) + (2^{i-1})c = \\ &\underset{\text{i=n}}{=} 2^n T(n) + (2^n - 1)c = \\ &= 2^n \times Costo(Stampa(B)) + (2^n - 1)c = \\ &= \Theta(2^n \times Costo(Stampa(B))) \end{aligned}$$

La complessità dell'algoritmo risulta quindi essere esponenziale in n. □

Esercizio 10 (Compitino 25/05/2016). 🏷 `Hashing` 🏷 `Alberi`
Si consideri un array S di n chiavi intere che possono assumere soltanto $d \leq n$ valori distinti.

1. Si dia il codice di un algoritmo che con un'unica scansione di S conti il numero di chiavi distinte che occorrono un numero *dispari* di volte in S. Usare un dizionario D inizialmente vuoto (non interessa l'implementazione di D);
2. Assumendo che D sia implementato come un albero AVL, si analizzi la complessità in funzione di n e del numero d di chiavi distinte.

Soluzione — Punto 1.

```
ContaDispari(S)
    D = nuovo dizionario vuoto
    count = 0
    for i = 1 to n
        u = Search(D, S[i])
        if u = NIL
            count = count + 1
            Insert(D, S[i])
        else
            count = count - 1
            Delete(D, u)
    return count
```

Si noti che ad ogni iterazione del ciclo `for`, il dizionario D non contiene duplicati: questo perché se viene incontrato un elemento già presente in D, esso verrà eliminato. In questo modo è anche possibile contare gli elementi dispari, perché l'esecuzione passerà dall'`else` solo per gli elementi presenti un numero pari di volte. In altre parole il dizionario D contiene in ogni momento solo le chiavi osservate un numero dispari di volte. □

Soluzione — Punto 2.

Quando il dizionario D è implementato con un albero AVL tutte le operazioni su di esso richiedono tempo logaritmico nel numero di elementi ivi contenuti. Osserviamo che il dizionario contiene al massimo d elementi, per cui le operazioni di dizionario hanno tutte costo $\mathcal{O}(\log d)$. Iterando questo procedimento per n volte otteniamo che la complessità totale di questo algoritmo è $T(n, d) = \mathcal{O}(n \log d)$. □

Esercizio 11 (Esame 16/05/2002). 🏷 `Hashing`

Si vuole inserire la sequenza di chiavi intere $19, 36, 6, 132, 39, 262, 45, 67, 64, 28, 15, 60$ in una tabella hash inizialmente vuota di lunghezza $m = 16$, usando l'indirizzamento aperto con scansione quadratica mediante la seguente funzione hash per ciascuna chiave k:

$$h(k, i) = \left(h'(k) + \frac{1}{2}i + \frac{1}{2}i^2\right) \mod m, \quad \text{dove } i = 0, 1, \ldots, \text{ e } h(k) = k \mod m.$$

Mostrare l'esecuzione delle inserzioni elencando le posizioni $h(k, i)$ esaminate nella tabella hash durante la scansione operata con ciascuna chiave k.

Soluzione.

Chiave k	Posizioni h(k, i) esaminate nella tabella
19	[3]
36	[4]
6	[6]
132	[4, 5]
39	[7]
262	[6, 7, 9]
45	[13]
67	[3, 4, 6, 9, 13, 2]
64	[0]
28	[12]
15	[15]
60	[12, 13, 15, 2, 6, 11]

□

Esercizio 12 (Esame 03/07/2017). `Complessità`
Sia dato un grafo G e un intero positivo k.

1. Progettare un algoritmo che stabilisce se esiste in G una clique di k nodi (un sottografo completo di k nodi) e valutarne la complessità. Commentare inoltre la scelta della rappresentazione in memoria del grafo utilizzata.
2. Questo problema è NP-completo? Motivare la risposta.

Soluzione — Punto 1.
Questo problema può essere risolto mediante l'utilizzo della procedura `GeneraBinarie`, che stabilirà di volta in volta quale sottografo controllare. La procedura `Elabora` quindi controllerà se vi sono tutti gli archi tra i k correntemente considerati.

```
Clique(G, k)
    S = nuovo array di |V| posizioni
    GeneraBinarie(G, S, 0) // Passiamo anche il grafo a GeneraBinarie
    /* Se arriviamo all'istruzione successiva l'algoritmo termina dichiarando che non esiste in G
        una clique di k nodi */
    *failure*
```

```
Elabora(G, S, k)
    num = 0
    for i = 1 to |V|
        if S[i] = 1
```

```
            num = num + 1
    if num ≠ k
        return // Il sottografo considerato non e' fatto di k nodi

    for i = 1 to |V|
        for j = i+1 to |V|
            if S[i] = 1 && S[j] = 1 // i e j appartengono al sottografo descritto da S
                if A[i, j] = 0
                    return // Non esiste l'arco (i,j)
    /* Se arriviamo all'istruzione successiva l'algoritmo termina dichiarando che esiste una
        clique di k nodi in G */
    *success*
```

Come rappresentazione del grafo in memoria è stata scelta la matrice di adiacenza (A nel codice) in modo da velocizzare le operazioni di controllo sulla presenza degli archi ($\Theta(1)$). La complessità di questa soluzione, poiché utilizza la procedura `GeneraBinarie`, è pari a:

$$T(|V|, |E|) = \mathcal{O}(2^{|V|}T(Elabora))$$
$$T(Elabora) = \mathcal{O}(|V|^2)$$
$$T(|V|, |E|) = \mathcal{O}(2^{|V|}|V|^2)$$

che è esponenziale nel numero di vertici del grafo. □

Soluzione — Punto 2.
Sì, il problema $CLIQUE$ è NP-completo in quanto

1. $CLIQUE \in NP$ (l'algoritmo che controlla se i k nodi forniti come input formano una clique richiede tempo polinomiale in k);
2. $SAT \leq_{\mathcal{P}} CLIQUE$ e SAT sappiamo essere NP-completo.

□

Esercizio 13 (Esame 12/06/2015). 🏷 `Programmazione Dinamica`
Sia data una sequenza L di n numeri interi distinti. Si scriva una procedura basata sulla programmazione dinamica per trovare la più lunga sottosequenza crescente di L (per esempio, se $L = 9, 15, 3, 6, 4, 2, 5, 10$, la più lunga sottosequenza crescente è $3, 4, 5, 10$).
Si fornisca la regola ricorsiva, lo pseudocodice della procedura e ricostruire la soluzione sulla sequenza di esempio. Si analizzi quindi la complessità in tempo e spazio dell'algoritmo.

Soluzione.
Notiamo, come prima cosa, che in questo caso la tabella di programmazione dinamica è in realtà un array. Questo è dovuto al fatto che si ha una sola sequenza su cui applicare l'algoritmo e non abbiamo altri parametri all'infuori della lunghezza della sottosequenza. La tabella di programmazione dinamica sarà quindi un array $M[1 \dots n]$: per ogni $i \in [1, n]$, $M[i]$ corrisponderà alla lunghezza della più lunga sottosequenza crescente di L contenuta in $L[1, \dots, i]$, che termina con $L[i]$. Per

questo motivo il sottoproblema elementare è quello per cui la lunghezza della sequenza è pari a 1, da cui $M[1] = 1$ (la sequenza lunga 1 è certamente crescente).
La regola ricorsiva è quindi definita come:

$$M[i] = 1 + \max_{1 \leq j \leq i} \{M[j] \mid L[j] < L[i]\}.$$

Secondo questa regola è possibile espandere la più lunga sottosequenza crescente con il valore $L[i]$ solo se questo è maggiore del massimo $L[j]$, con $j < i$ (in questo modo infatti aggiungendo $L[i]$ alla soluzione otteniamo una sequenza ancora crescente).

```
1  LIS_Length(L)
2      M = nuovo array di n posizioni
3      M[1] = 1
4      for i = 2 to n
5          max = 0
6          for j = 1 to i-1
7              if L[i] > L[j] && M[j] > max
8                  max = M[j]
9          M[i] = 1 + max
10
11     pos_max = 1
12     for i = 2 to n
13         if M[i] > M[pos_max]
14             pos_max = i
15     return M[pos_max]
```

La soluzione globale sarà data dal valore massimo di M.
La complessità di questa soluzione è $T(n) = \Theta(n^2)$ in quanto, ad ogni iterazione, si effettua una ricerca del massimo nella porzione precedente dell'array e le iterazioni sono n. Lo spazio aggiuntivo, dovuto all'array di programmazione dinamica, è pari a $S(n) = \Theta(n)$.
Per ricostruire la soluzione bisogna cercare, da destra verso sinistra, i vari massimi contenuti in M secondo la seguente procedura.

```
1  LIS(L, M)
2      max = -1
3      for i = 1 to n
4          if M[i] > max
5              max = M[i]
6      for i = n down to 1
7          if M[i] = max
8              print L[i]
9              max = max - 1
```

Sulla sequenza di esempio il risultato è dato da (in grassetto i valori che soddisfano la condizione dell'ultimo `if`):

$$\begin{array}{rcccccccc} L = & 9 & 15 & \mathbf{3} & 6 & \mathbf{4} & 2 & \mathbf{5} & \mathbf{10} \\ M = & 1 & 2 & \mathbf{1} & 2 & \mathbf{2} & 1 & \mathbf{3} & \mathbf{4} \end{array}$$

□

Esercizio 14 (Esame 12/06/2015 - II). 🏷 `Alberi`

Sia T un albero binario di ricerca contenente n chiavi reali distinte. Progettare un algoritmo *non ricorsivo* che determini la più piccola chiave positiva presente in T, e analizzarne la complessità. Se in T non esistono chiavi positive, l'algoritmo deve restituire la stringa `'no positive key'`.

Soluzione.

```
1  MinPositivo(T)
2      u = T.root
3      min = ∞
4      while u ≠ NIL
5          if u.key > 0
6              min = u.key
7              u = u.left
8          else
9              u = u.right
10     if min ≠ ∞
11         return min
12     else
13         return 'no positive key'
```

L'algoritmo effettua una ricerca classica del minimo in un albero binario di ricerca (scendendo quindi a sinistra) avendo però l'accortezza di registrare il valore minimo solo se questo è positivo. Se la chiave contenuta in un nodo è negativa, la visita prosegue a destra in quanto certamente a sinistra ci saranno soltanto altri valori negativi. Poiché l'algoritmo esegue una visita dell'albero dalla radice verso le foglie, spendendo tempo costante fuori dal ciclo e nel corpo di esso, la complessità di questo algoritmo è proporzionale all'altezza dell'albero ovvero $T(n) = O(h) = O(n)$. □

Esercizio 15 (Esame 05/09/2017). 🏷 `Alberi`

Dato un albero binario T in cui ogni nodo contiene un intero positivo, diciamo che la *media* di un sottoalbero non vuoto è la somma degli interi contenuti nei suoi nodi diviso il numero di tali nodi; se il sottoalbero è vuoto, la media si assume essere pari a zero.
Scrivere lo pseudocodice di un algoritmo ricorsivo che, presa in ingresso la radice di T, restituisce *la radice del sottoalbero* la cui media è massima rispetto a quella degli altri sottoalberi, e se ne valuti la complessità in tempo al caso pessimo (si assuma che il sottoalbero di media massima sia unico).

Soluzione.

Per poter risolvere questo problema con un'unica procedura ricorsiva è necessario restituire, ad

ogni chiamata, più informazioni di quelle richieste dal testo dell'esercizio. In particolare dovremo ottenere i seguenti valori:

- `dim`: la dimensione del sottoalbero radicato nel nodo corrente (incluso il nodo stesso);
- `sum`: la somma dei valori nel sottoalbero radicato nel nodo corrente, più il valore del nodo corrente;
- `maxM`: media massima tra tutti i sottoalberi dei nodi discendenti dal nodo corrente;
- `maxNode`: radice del nodo di media `maxM`.

Lo pseudocodice è il seguente:

```
1  MediaMax(u)
2      if u = NIL
3          return <0, 0, 0, NIL> // I valori sono ordinati come sopra
4      <dimSx, sumSx, maxMediaSx, maxNodeSx> = MediaMax(u.left)
5      <dimDx, sumDx, maxMediaDx, maxNodeDx> = MediaMax(u.right)
6
7      dim = dimSx + dimDx + 1 // Include il nodo u
8      sum = SumSx + SumDx + u.key // Include il nodo u
9
10     if sum/dim > max(maxMediaSx, maxMediaDx)
11         maxNode = u
12         maxMedia = sum / dim
13     else
14         maxMedia = max(maxMediaSx, maxMediaDx)
15         if maxMedia = maxMediaSx
16             maxNode = maxNodeSx
17         else
18             maxNode = maxNodeDx
19
20     return <dim, sum, maxMedia, maxNode>
```

Questa soluzione effettua una postvisita dell'albero, effettuando del lavoro costante al di fuori delle chiamate ricorsive. Per cui $T(n) = \Theta(n)$. □

Esercizio 16 (Esame 05/09/2017 - II). 🏷 `Grafi`

Dato un grafo non orientato e connesso $G = (V, E)$, un arco (u, v) è detto *ponte* se la sua rimozione divide il grafo in due componenti connesse. Scrivere lo pseudocodice di un algoritmo che trova, se esiste, un ponte in G, e se ne valuti la complessità in tempo al caso pessimo.

Soluzione.

Esistono tre modi per risolvere il problema, la prima strategia è quella di rimuovere a turno un arco dal grafo e controllare se così facendo si sono ottenute due componenti connesse, per fare ciò occorre eseguire una BFS su di $G' = (V, E \setminus (u, v))$ per ogni arco $(u, v) \in E$ e se la BFS su di G' non raggiunge tutti i nodi possiamo concludere che l'arco (u, v) è un ponte.

```
CheckConnesso(G, u, v) // Controlla se G - (u,v) e' connesso
    for each v ∈ V
        v.color = bianco
    s = any v ∈ V // Scelgo un nodo sorgente a caso
    s.color = grigio

    Q = ∅ // Coda vuota
    Enqueue(Q, s)

    while Q ≠ ∅ // BFS
        x = Dequeue(Q)
        for each y ∈ Adj[x]
            if (x ≠ u && y ≠ v && y.color = bianco) // Controllo sull'arco (u,v) rimosso
                y.color = grigio
                Enqueue(Q, y)
        x.color = nero

    connesso = true // Controllo se il grafo G - (u,v) e' connesso
    for each v ∈ V
        if v.color = bianco
            connesso = false
            break

    return connesso

Ponte(G) // Controlla se in G c'e' un ponte
    for each (u,v) ∈ E
        if !CheckConnesso(G, u, v)
            return (u,v)
    return NIL
```

La complessità della soluzione sopra esegue una BFS per ogni arco, da cui

$$T(|V|, |E|) = T(n, m) = \mathcal{O}(m(n + m)).$$

Una seconda soluzione di complessità $\mathcal{O}(n(n + m))$ nasce dall'osservazione che se un ponte (u, v) esiste, allora tale arco apparterrà all'albero di copertura di G ottenuto con una BFS (o DFS), per cui possiamo restringere l'insieme degli archi da rimuovere uno ad uno a quelli in E', dove $G' = (V, E')$ è un albero di copertura ottenuto tramite una visita:

```
Ponte(G) // Controlla se in G c'e' un ponte
    G1 = (V, E1) = AlberoCopertura(G) // Si fa con una BFS

    for each (u,v) ∈ E1
        if !CheckConnesso(G, u, v)
            return (u,v)
    return NIL
```

Si noti che G' è un albero, da cui $|E'| = |V| - 1 = n - 1$, segue la complessità:

$$T(n, m) = \Theta(n + m) + (n - 1) \cdot \mathcal{O}(n + m) = \mathcal{O}(n(n + m)).$$

Esiste inoltre una terza soluzione, ottimale in $\Theta(n + m)$ che si basa sull'osservazione del fatto che eventuali archi ponte non fanno parte di alcun ciclo. □

Esercizio 17 (Esame 05/09/2017 - III). 🏷 `Complessità`

Il problema *Exact Cover* richiede di stabilire se, data una collezione di insiemi $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ ciascuno sottoinsieme dell'intervallo $[1, n]$ dei primi n interi positivi, esiste una famiglia di sottoinsiemi di $\mathcal{S}$, a due a due disgiunti e tali che la loro unione è uguale a tutto l'intervallo $[1, n]$.
Scrivere lo pseudocodice di una procedura che verifica se una data famiglia di insiemi $\mathcal{S}^* \subseteq \mathcal{S}$ è un *exact cover* per $[1, n]$ e se ne valuti la complessità in tempo al caso pessimo. Si assuma che la procedura riceva in input una matrice binaria $m \times n$ tale che $S[i, j] = 1$ se e solo se l'insieme S_i contiene l'intero positivo j, e un vettore binario $V[1 \ldots m]$ tale che $V[k] = 1$ se e solo se il sottoinsieme S_k fa parte della famiglia $\mathcal{S}^*$.

Soluzione.

Data una famiglia $\mathcal{S}^*$ di sottoinsiemi, abbiamo che $\mathcal{S}^*$ è una exact cover se e solo se

$$\forall j \in [1, n] \rightarrow |\{S_i \in \mathcal{S}^* | j \in S_i\}| = 1.$$

Ossia ogni valore nell'intervallo $[1, n]$ è contenuto in esattamente un $S_i \in \mathcal{S}^*$, una procedura di verifica è quindi:

```
1  Elabora(V, S, n, m) // Controlla se V e' una exact cover
2      for j = 1 to n
3          conta = 0
4          for each S_i ∈ S* // i tale che V[i] == 1
5              if S[i][j] = 1
6                  conta = conta + 1 // Conto a quanti insiemi appartiene j
7          if conta ≠ 1 // j non appartiene ad esattamente 1 sottoinsieme
8              return false
9      return true
```

La complessità della funzione riportata è quindi $\mathcal{O}(n \cdot |\mathcal{S}^*|)$ che al caso pessimo è $\mathcal{O}(nm)$. Se volessimo ricavare una soluzione $\mathcal{V}^*$ al problema data la matrice S, possiamo utilizzare la procedura `GeneraBinarie` per generare tutti i possibili sottoinsiemi di $\mathcal{S}$ e verificare se ognuno di essi è soluzione applicandovi `Elabora`. La complessità del trovare una soluzione al problema risulterebbe quindi essere $\mathcal{O}(2^m \times T_{elabora}(n, m))$, esponenziale in m. □

Esercizio 18 (Esame 03/07/2015). 🏷 `Programmazione Dinamica`

Si consideri il seguente problema (SUBSETSUM): Dato un insieme di n interi positivi $A = \{a_1, \ldots, a_n\}$, determinare se esiste un sottoinsieme $A' \subseteq A$ tale che la somma degli elementi in A' abbia valore k. Si consideri una possibile soluzione del problema del SUBSETSUM con il metodo della Programmazione Dinamica. Si descriva a parole la soluzione complessiva, ma si riporti esplicitamente la regola ricorsiva.

Soluzione.
Possiamo considerare come sottoproblema P_j quello di determinare se esiste un sottoinsieme dei primi i elementi di A che ha somma pari a j. La tabella di programmazione dinamica è quindi una matrice di dimensione $(n+1) \times (k+1)$ e contiene valori booleani ($true$ / $false$).
I sottoproblemi elementari sono quindi:

- $M[0,0] = true$ (gli elementi dell'insieme vuoto sommano a zero);
- $M[0,j] = false$ per ogni $j = 1 \dots k$ (su un insieme vuoto non esistono sottoinsiemi di somma j).

La regola ricorsiva, ottenuta dalla considerazione sul problema generico P_j, è dunque:

$$M[i,j] = \begin{cases} true, & \text{se } M[i-1,j] = true \\ true, & \text{se } j \geq A[i] \text{ e } M[i-1, j-A[i]] = true \\ false, & \text{altrimenti} \end{cases} .$$

Possiamo dunque concludere che A contiene un sottoinsieme di somma pari a k se $M[n,k] = true$ e la complessità di questa soluzione è $T(n,k) = T(nk)$. Si noti che questa complessità è di tipo pseudopolinomiale in k, in quanto k rappresenta un valore intero e non una dimensione dell'input. □

Esercizio 19 (Esame 13/07/2010). 🏷 `Hashing`
Si consideri la gestione di tabelle hash a indirizzamento aperto mediante scansione lineare.

1. Definire la formula da utilizzare per la scansione lineare per una data funzione hash $h_{aux}(k)$.
2. Scrivere lo pseudocodice per il seguente inserimento di una chiave k: se la posizione correntemente esaminata risulta occupata da un'altra chiave k', allora k prende il posto di k' in tale posizione e si continua con l'inserimento di k' a partire dalle posizioni successive (in modo circolare).
3. Dimostrare che la dimensione dei *cluster* primari (agglomerati) così ottenuti non cambia rispetto all'inserimento standard visto a lezione.

Soluzione — Punto 1.
Posto m dimensione della tabella, la formula da utilizzare per la scansione lineare è:

$$h(k,i) = (h_{aux}(k) + i) \mod m$$

□

Soluzione — Punto 2.

```
1 Hash_Insert(T, k)
2     i = 0
3     do
```

```
        j = j(k, i)
        if T[j] = NIL || T[j] = DELETED
            /* Posizione libera, termina subito */
            T[j] = k
            return j
        else
            /* Scambia T[j] con k e va avanti con l'inserimento */
            tmp = T[j]
            T[j] = k
            k = tmp
            i = i + 1
    while i ≠ m
    error "Overflow della tabella hash"
```

□

Soluzione — Punto 3.
Si osservi che la funzione hash rimane invariata rispetto all'inserimento classico, per cui gli insiemi delle chiavi collidenti tra di loro sono invariati. Segue da ciò il fatto che un inserimento occupa una posizione aggiuntiva al termine della sequenza agglomerata pre-esistente, in modo analogo a quanto accade con l'inserimento standard, da cui la tesi: la lunghezza dei cluster primari rimane invariata. □

Esercizio 20. 🏷 `Complessità`
Dato un grafo $G = (V, E)$, si definisce *Cammino Hamiltoniano* un cammino che tocca ogni nodo del grafo esattamente una volta. Il problema Hamiltonian Path chiede, dato G, se questo contiene o meno un cammino hamiltoniano. Si fornisca una riduzione di questo problema a SAT.

Soluzione.
Supponiamo che $V = \{1, 2, \ldots, n\}$ e definiamo un *cammino* di lunghezza t su G come una sequenza di t nodi $v \in V$, ovvero $c = (c_1, c_2, \ldots, c_t)$. Un cammino Hamiltoniano è quindi un cammino lungo n, ed è composto da una permutazione π degli elementi di V. Diremo quindi che $\pi(i) = j$ se l'i-esimo nodo nel cammino è il nodo j, e vogliamo trovare una permutazione π tale che

$$(\pi(i), \pi(i+1)) \in E \ \forall i = 1, 2, \ldots, n-1.$$

Dato G costruiamo un'istanza di SAT, $R(G)$ nel modo seguente. $R(G)$ avrà $|V|^2$ variabili booleane, ciascuna chiamata x_{ij} con $0 \leq i, j \leq n$, col significato che $\forall (i, j) : (x_{ij} = True) \rightarrow (\pi(i) = j)$ cioè che la posizione i nel cammino hamiltoniano è occupata dal nodo j. La corrispondente formula istanza di SAT viene quindi costruita facendo l'`and` booleano tra tutte le seguenti clausole.

1. Ogni nodo j di G deve apparire nel cammino Hamiltoniano:
   - $x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj} \quad \forall j$
2. Ogni nodo j di G appare esattamente una volta nel cammino Hamiltoniano:

- $\neg x_{ij} \vee x_{kj} \quad \forall i, j, k$ tali che $i \neq k$

3. Ogni posizione i nel cammino deve essere "occupata", cioè deve avere un nodo corrispondente:

   - $x_{i1} \vee x_{i2} \vee \cdots \vee x_{in} \quad \forall i$

4. Dati i nodi $j \neq k$, questi non devono occupare la stessa posizione nel cammino:

   - $\neg x_{ij} \vee x_{ik} \quad \forall i, j, k$ tali che $j \neq k$

5. Due nodi i e j non adiacenti nel grafo non possono essere adiacenti nel cammino:

   - $\neg x_{ki} \vee \neg x_{k+1,j} \quad \forall (i, j) \notin E, k = 1, 2, \ldots, n-1$.

L'and logico di tutte le clausole generate tramite le regole precedenti fornirà l'istanza di SAT che sarà soddisfacibile *se e solo se* G contiene un cammino hamiltoniano. Di seguito forniamo una dimostrazione intuitiva della correttezza di questa riduzione.

Dimostrazione ($SAT \to HAM$). Posti $n = |V|$ ed $m = |E|$, $R(G)$ contiene $\mathcal{O}(n^3)$ clausole ed è ottenibile in tempo polinomiale in n (basta generare tutte le clausole seguendo le regole).

Supponiamo che la formula così ottenuta sia soddisfacibile: dalle prime due regole esiste una sola posizione i per ogni nodo j tale che $x_{ij} = True$. Analogamente, dalle regole 3 e 4 possiamo ricavare che esiste un solo nodo j per ogni posizione i tale che $x_{ij} = True$. Quindi nella permutazione π vale $\pi(i) = j$ se e solamente se $x_{ij} = True$. La regola 5, infine, ci garantisce che $(\pi(1), \pi(2), \ldots, \pi(n))$ sia effettivamente un cammino Hamiltoniano. Segue che se la formula ottenuta è soddisfacibile, G ha un cammino Hamiltoniano. □

Dimostrazione ($HAM \to SAT$). Supponiamo adesso che G abbia un cammino Hamiltoniano e dimostriamo che allora la formula di SAT è soddisfacibile.

Sia $(\pi(1), \pi(2), \ldots, \pi(n))$ un cammino Hamiltoniano in G. Un assegnamento di $x_{ij} = True$ per ogni i e j tali che $\pi(i) = j$ soddisfa chiaramente la formula corrispondente di SAT. □

Con questo abbiamo dimostrato che un grafo G ha un cammino hamiltoniano se e solamente se la corrispondente formula istanza di SAT è soddisfacibile, mentre se G non ha un cammino hamiltoniano allora non esiste un assegnamento delle variabili nella corrispondente formula che la renda vera. Questo conclude la riduzione.

Si noti infine che questa riduzione ci consente di dire che SAT è almeno difficile quanto Hamiltonian Path, ma non ci dice niente sull'eventuale NP-Completezza di quest'ultimo. □

Esercizio 21 (Esame 22/06/2004). ???

Il signor Caio possiede una cisterna di capacità C litri e la vuole usare per consegnare al cugino Sempronio almeno L litri di vino. La sua produzione di vino è suddivisa in n botti numerate da 1 a n e di capacità $c_1, c_2, \ldots, c_n$ litri ciascuna. Il vino va travasato dalle botti alla cisterna per la consegna. Progettare due algoritmi che identificano quali botti da utilizzare nel carico, e valutarne la complessità,

1. assumendo che ogni botte possa essere utilizzata parzialmente quando viene versata nella cisterna;
2. assumendo che ogni botte **non** possa essere utilizzata parzialmente quando viene versata nella cisterna; ovvero, utilizzando la botte i si versano tutti i c_i litri di vino della cisterna (chiaramente solo se $c_i \leq C$).

Commentare gli algoritmi proposti.

Suggerimento.
Evidentemente nel 2004 non esistevano ancora gli *zaini* impermeabili... □