Complementi di Algoritmi e Strutture Dati

(III anno Laurea Triennale - a.a. 2016/17)

Prova scritta 19 giugno 2017

NB: I punteggi sono indicativi.

Esercizio 1 - Tabelle hash Supponiamo di usare una tabella hash di $m = 11$ caselle (numerate da 0 a 10) e rehashing con la seguente famiglia di funzioni:

$$f(k, i) = (k + i) \mod m$$

Supponiamo inoltre che la tabella abbia inizialmente occupati i posti di indice $1, 4, 5, 6$ (con elementi di chiave $11, 4, 15, 6$) e liberi gli altri posti. Graficamente, la situazione è questa:

indice	0	1	2	3	4	5	6	7	8	9	10
contenuto		11			4	15	6				

1. Inserire nella tabella l'elemento di chiave $k = 16$. Dire se ci sono collisioni (in caso affermativo dire quali collisioni ci sono) e mostrare la tabella dopo l'inserimento.

   SOLUZIONE: $f(16, 0) = 5$: collide; $f(16, 1) = 6$: collide; $f(16, 2) = 7$: casella vuota e inserisco.

indice	0	1	2	3	4	5	6	7	8	9	10
contenuto		11			4	15	6	16			

2. Successivamente, cancellare dalla tabella l'elemento di chiave $k = 15$. Dire quali caselle vengono accedute e mostrare la tabella dopo la cancellazione.

   $f(15, 0) = 4$: nella casella c'è elemento di chiave $4 \neq 15$ perciò proseguo la ricerca; $f(15, 1) = 5$: nella casella c'è elemento di chiave $= 15$ e lo cancello inserendo marchio.

indice	0	1	2	3	4	5	6	7	8	9	10
contenuto		11			4	X	6	16			

3. Successivamente, eseguire una ricerca con chiave $k = 26$. Dire quali caselle vengono accedute e dire il risultato della ricerca (trovato oppure non trovato).

   $f(26, 0) = 4$: casella occupata da elemento di chiave $4 \neq 26$, continuo; $f(26, 1) = 5$: casella marcata, continuo; $f(26, 2) = 6$: casella occupata da elemento di chiave $6 \neq 26$, continuo; $f(26, 3) = 7$: casella occupata da elemento di chiave $16 \neq 26$, continuo; $f(26, 4) = 8$: casella vuota, elemento non trovato.

4. Inserire infine un elemento di chiave $k = 26$. Dire se ci sono collisioni, nel caso quali, e mostrare la tabella dopo l'inserimento.

   $f(26, 0) = 4$: collide; $f(5, 1) = 5$: casella marcata, inserisco qui.

indice	0	1	2	3	4	5	6	7	8	9	10
contenuto		11			4	26	6	16			

Nota bene: ogni operazione va eseguita sul risultato delle operazioni precedenti.

Esercizio 2 - Sorting Consideriamo una versione di Quicksort che lavora su una lista linkata s invece che su un array.
Sulla lista supponiamo di avere, implementate con costo costante, le classiche operazioni:
• *isEmpty()* – test se vuota,
• *deleteFirst()* – ritorna primo elemento cancellandolo,
• *addToEnd(int x)* – inserisce elemento x in fondo.
La scelta del pivot consiste nel prendere (togliendola) la testa della lista s:
    $p = \text{deleteFirst}(s)$;
Il partizionamento consiste nello scorrere (quel che rimane del)la lista s ed ogni volta trasferire la testa in s_1 se minore del pivot e in s_2 se maggiore o uguale:
    // dalla lista s ho già tolto il pivot p
    s_1 = new list(); // qui metterò elementi $< p$
    s_2 = new list(); // qui metterò elementi $\geq p$
    while (NOT s.isEmpty())
        $x = s$.deleteFirst();
        if $x < p$ s_1.addToEnd(x);
        else s_2.addToEnd(x);
Ovviamente vengono poi eseguite le chiamate ricorsive su s_1 e s_2 e la lista risultato (ordinata) è la concatenazione $s_1 \cdot p \cdot s_2$.
Si chiede di:

1. Far vedere che il partizionamento termina ed è corretto.

   POST: (elementi di s_1 tutti $< p$) AND (elementi di s_2 tutti $\geq p$) AND (unione elementi di s_1, s_2 = elementi di s iniziali).

   INV: (elementi di s_1 tutti $< p$) AND (elementi di s_2 tutti $\geq p$) AND (unione elementi di s, s_1, s_2 = elementi di s iniziali).

   INV vale all'inizio del ciclo? banalmente sì essendo s_1, s_2 vuote.

   INV si mantiene? sì perchè l'elemento tolto da s viene trasferito in s_1 o s_2, ed esattamente in s_1 se $< p$ e in s_2 se $\geq p$.

   All'uscita del ciclo vale POST? banalmente sì perché INV AND (s vuota) coincide con POST.

   Termina perché la lunghezza di s decresce ad ogni giro ed è limitata inferiormente da 0.

2. Questo partizionamento ha complessità tempo lineare in n (lunghezza della lista)? Perchè? Che cosa si può dire riguardo alla complessità spazio?

   Tempo: lineare perché il ciclo viene eseguito una volta per ogni elemento originariamente in s (che viene tolto), ovvero n volte, e le operazioni fatte ad ogni giro hanno costo costante.

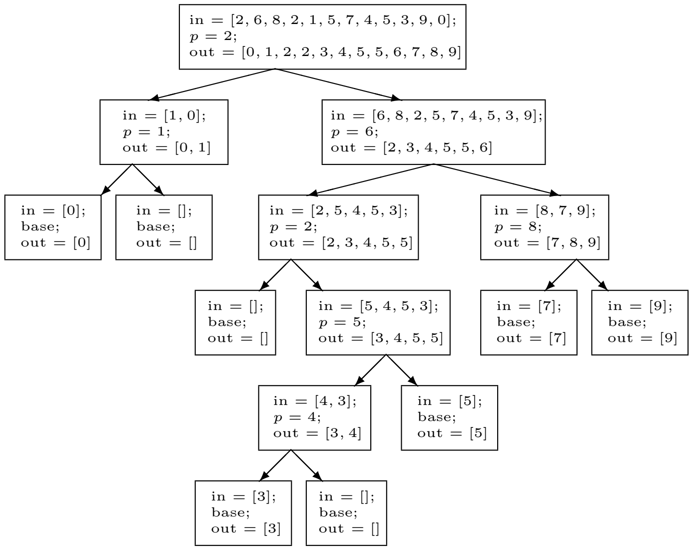
   Spazio: lo spazio aggiuntivo richiesto dal partizionamento è $O(1)$ perché gli elementi sono man mano tolti da s e aggiunti ad una tra s_1, s_2, quindi in ogni momento il numero totale di caselle, nelle tre liste, è sempre n.

3. Simulare l'esecuzione dell'intero algoritmo **Quicksort** (in questa versione) su una lista linkata che contiene nell'ordine gli elementi

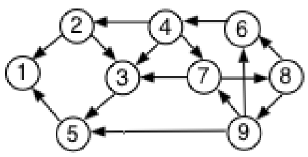
   $$[2, 6, 8, 2, 1, 5, 7, 4, 5, 3, 9, 0].$$

   Disegnare l'albero delle chiamate ricorsive. Per ogni chiamata indicare:

   - la lista di input
   - se non si tratta di un caso base (foglia nell'albero delle chiamate) indicare il pivot p (si intende che le due sotto-liste s_1, s_2 saranno le liste di input dei due nodi figli sinistro e destro).
   - la lista risultato (parziale)



Esercizio 3 - Grafi Si esegua, sul seguente grafo:



l'algoritmo per il calcolo delle componenti connesse. In particolare, si diano:

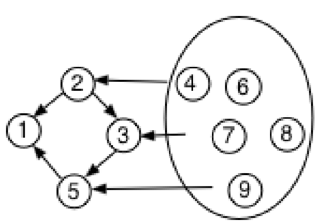
- i tempi di inizio e fine visita ottenuti per ogni nodo in seguito alla visita in profondità

  8: 1/18 6: 2/15 4: 3/14 2: 4/11 1: 5/6 3: 7/10 5: 8/9 7: 12/13 9: 16/17

- la sequenza delle componenti fortemente connesse `Ord`$^{\leftrightarrow}$ ottenuta

  {8, 7, 4, 6, 9} {2} {3} {5} {1}

- il grafo quoziente.



Esercizio 4 - Design e analisi di algoritmi Un distributore di bibite contiene al suo interno n monete i cui valori (interi positivi) sono memorizzati in un array $c[1..n]$. Si consideri il problema di decidere se è possibile erogare un resto esattamente uguale a R (intero positivo) utilizzando un opportuno sottoinsieme delle n monete a disposizione. Sia $P(k, r)$ $(1 \leq k \leq n, 0 \leq r \leq R)$ il sottoproblema di decidere se è possibile erogare un resto esattamente uguale a r utilizzando le monete $\{1, \ldots, k\}$. Si noti che, a differenza del problema del resto visto a lezione, non è necessario erogare il resto con il numero minimo di monete.

1. Dare una definizione induttiva di $P(k, r)$, giustificandone la correttezza.

Base

$P(1,r) = \begin{cases} T & \text{se } r = 0 \text{ oppure } r = c[1] \\ F & \text{altrimenti} \end{cases}$

Con una sola moneta possiamo solo erogare resto 0, oppure un resto esattamente uguale al valore della moneta.

Passo induttivo

per $k > 1$, $P(k,r) = \begin{cases} P(k-1, r-c[k]) \vee P(k-1,r) & \text{se } c[k] \leq r \\ P(k-1,r) & \text{se } c[k] > r \end{cases}$

Con $k > 1$ monete, se il valore della k-sima moneta è minore o uguale del resto richiesto, abbiamo due possibilità: usare la k-sima moneta e poi erogare con le restanti monete il resto residuo, oppure non usarla ed erogare il resto richiesto con le altre. Se invece il valore della k-sima moneta è maggiore del resto richiesto non possiamo usarla, quindi ci rimane soltanto la seconda possibilità.

2. Descrivere un algoritmo di programmazione dinamica per risolvere il problema.

```
for (r = 0; r <= R; r++) P(1,r) = (r=0 || r=c[1])

for (k = 2; k <= n; k++)
  for (r = 0; r <= R; r++)
    if (c[k]<= r) P(k,r) = P[k-1,r-c[k]) || P[k-1,r]
    else P(k,r) = P(k-1,r)
```

3. Determinare il costo computazionale dell'algoritmo descritto, motivando la risposta.
   Il costo computazionale è (ovviamente) $O(n \cdot R)$.

4. Spiegare cone modificare l'algoritmo per determinare anche quali sono le monete da erogare.
   Per determinare anche quali sono le monete da erogare possiamo utilizzare un'ulteriore matrice booleana $n \times R$, sia U, tale che $U[k,r] = T$ se e solo se utilizzo la k-sima moneta per dare il resto r.

```
//assumiamo la matrice inizializzata con tutti false
for (r = 0; r <= R; r++)
  if (r=c[1]) U[1,r] = true

for (k = 2; k <= n; k++)
  for (r = 0; r <= R; r++)
    if (c[k]<= r && P[k-1,r-c[k]) U[k,r] = true
```

Esercizio 5 - NP-completezza Si consideri il seguente problema TWO-CLIQUE: dato un grafo G e due interi $h, k > 0$, determinare se G contiene due clique disgiunte di dimensione rispettivamente h e k.

1. Provare che TWO-CLIQUE è in NP.

   Un certificato per il problema TWO-CLIQUE è formato da una coppia di sottoinsiemi di nodi. È facile verificare in tempo polinomiale che i due sottoinsiemi siano disgiunti, siano delle clique e abbiano dimensione rispettivamente h e k.

2. Provare che TWO-CLIQUE è NP-hard.

   Diamo una riduzione da CLIQUE in TWO-CLIQUE. Un'istanza del problema CLIQUE è formata da un grafo G e un intero h. Trasformiamo (G, h) nell'istanza del problema TWO-CLIQUE formata da un grafo G' ottenuto aggiungendo a G un nodo isolato, sia u, e dagli interi h,1. La trasformazione è banalmente polinomiale e si ha ovviamente che (G, h) è una soluzione di CLIQUE (ossia, G ha una clique di dimensione h) se e solo se $(G', h, 1)$ ha due clique disgiunte di dimensione rispettivamente h e 1.