Esercizi del corso di Algoritmi e strutture dati

LT di Informatica - Università di Torino

Andras Horvath, Ugo de'Liguoro

27 maggio 2019

Indice

Capitolo 1

Correttezza: induzione ed invarianti di ciclo

Esercizio 1. Si consideri il seguente algoritmo che calcola il quadrato del suo parametro:

```
SQUARE(z)
1:      ▷ Pre: z numero intero ≥ 0
2:      ▷ Post: ritorna l'intero z^2
3: x ← 0
4: y ← 0
5: while x < z do
6:     y ← y + 2x + 1
7:     x ← x + 1
8: return y
```

Si dimostri formalmente la correttezza della funzione, cioè:

- si trovi l'invariante del ciclo,
- si dimostri che l'invariante è vero inizialmente e viene mantenuto dal ciclo
- usando l'invariante si dimostri che il valore restituito è z^2.

Esercizio 2. (**Esponenziale veloce**) Si consideri il seguente problema:

Ingresso: $x \neq 0$ reale, $n \geq 0$ intero.

Uscita: il reale x^n.

Si scrivano una versione ricorsiva ed una iterativa dell'algoritmo che calcola x^n sulla base delle seguenti equazioni:

$$x^{2k} = x^k \cdot x^k, \qquad x^{2k+1} = x^k \cdot x^k \cdot x.$$

Suggerimento: si basi la versione iterativa sull'invariante $x^n = y^k \cdot z$, dove y, k, z sono variabili di programma.

Esercizio 3. (**Ricerca lineare**) Si consideri il seguente problema:

Ingresso: $A[i..j]$ un array di interi, un intero a.

Uscita: $k \in i..j$ tale che $A[k] = a$ se esiste; -1 altrimenti.

Si scrivano una versione ricorsiva ed una iterativa dell'algoritmo che ritorna il minimo indice $k \in i..j$ tale che $A[k] = a$ se esiste; ritorna -1 altrimenti. Gli algoritmi devono essere corredati di pre e post-condizioni e rispettivamente degli asserti dell'ipotesi induttiva e dell'invariante di ciclo necessari per dedurrne la correttezza.

Esercizio 4. (**Ricerca dicotomica**) Si consideri il seguente problema:

Ingresso: $A[i..j]$ un array di interi ordinato in senso non decrescente, un intero a.

Uscita: $k \in i..j$ tale che $A[k] = a$ se esiste; -1 altrimenti.

Si scrivano una versione ricorsiva ed una iterativa dell'algoritmo di ricerca dicotomica che ritorna $k \in i..j$ tale che $A[k] = a$ se esiste; ritorna -1 altrimenti. Gli algoritmi devono essere corredati di pre e post-condizioni e rispettivamente degli asserti dell'ipotesi induttiva e dell'invariante di ciclo necessari per dedurrne la correttezza.

Esercizio 5. (**Bubble-Sort**, Cormen prob. 2-2) Il BUBBLE-SORT è un algoritmo di ordinamento che si basa sullo scambio di elementi adiacenti che non siano in ordine:

```
BUBBLE-SORT(A)
1: for i ← 1 to length(A) − 1 do
2:     for j ← length(A) downto i + 1 do
3:         if A[j] < A[j − 1] then
4:             scambia A[j] con A[j − 1]
5: return A
```

Sia $A' = \text{BUBBLE-SORT}(A)$; per provarne la correttezza dobbiamo stabilire che

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n]$$

dove $n = length(A) = length(A')$. Si stabilisca la correttezza dell'algoritmo attraverso i seguenti passi:

1. Si stabilisca con precisione l'invariante del **for** più interno, linee 2-6.
2. Usando la condizione di terminazione e l'invariante di cui al punto precedente, si enunci e si provi l'invariante del **for** più esterno, linee 1-7.

Esercizio 6. (**Regola di Horner**, Cormen prob. 2-3) Lo pseudocodice che segue implementa la regola di Horner per valutare un polinomio P di grado n in un punto x

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots))$$

dati i coefficienti $a_0, a_1, \ldots, a_n$:

```
1: y ← 0
2: for i ← n downto 0 do
3:     y ← a_i + x · y
```

Usando l'invariante

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

dedurre la post-condizione $y = \sum_{k=0}^{n} a_k x^k$. Quindi si confronti la regola di Horner con l'algoritmo ingenuo che calcola i singoli termini $a_i x^i$ prima di sommarli tra loro, per concludere che con la regola di Horner si effettua il minor numero possibile di moltiplicazioni. Sapreste proporre una soluzione intermedia come numero di moltiplicazioni tra quella di Horner e quella ingenua?

Capitolo 2

Crescita asintotica delle funzioni

Esercizio 1. Si dimostri che:

1. $3^{n-2} \in O(\pi^n)$
2. $\log^2 n \in o(\sqrt{n})$

Esercizio 2. Si forniscano due funzioni $f(n)$ e $g(n)$ tali che $f(n) \in \Omega(g(n))$ e $f(n) < g(n)$ per ogni $n \geq 1$.

Esercizio 3. Si dimostri che la seguente implicazione è falsa:

$$f_1(n) \in O(g_1(n)) \vee f_2(n) \in O(g_2(n)) \quad \Longrightarrow \quad f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$$

Esercizio 4. Si stabilisca se le funzioni 2^{n+1}, 2^{2n} e 4^n siano $O(2^n)$ provando o refutando l'asserto.

Esercizio 5. Sia $2^{O(\log n)}$ la classe di tutte le funzioni f da naturali a reali positivi tali che esista una costante reale $c > 0$ ed un n_0 tale che per ogni $n \geq n_0$ si abbia $f(n) \leq 2^{c \log_2 n}$.

Si dimostri che $O(n \log n) \neq 2^{O(\log n)}$ esibendo una funzione che appartenga ad uno dei due insiemi, ma non all'altro. Si stabilisca poi se siano inclusi uno nell'altro oppure se siano inconfrontabili per inclusione, giustificando le risposte.

Esercizio 6. Si dimostri per induzione su h che $\sum_{i=1}^{n} i^h = O(n^{h+1})$.

Esercizio 7. Siano $f(n), g(n)$ funzioni asintoticamente non negative; si dimostri che

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \ell > 0 \implies f(n) \in \Theta(g(n)).$$

Capitolo 3

Ordinamento in tempo quadratico

Esercizio 1. Si considerino gli algoritmi INSERTION-SORT e SELECT-SORT:

```
INSERTION-SORT(A)
for i ← 2 to length(A) do
    key ← A[i]
    j ← i − 1
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← key
```

```
SELECT-SORT(A)
for i ← 1 to length(A) − 1 do
    k ← i
    for j ← i + 1 to length(A) do
        if A[k] > A[j] then
            k ← j
    scambia A[i] con A[k]
return A
```

1. Si stabiliscano il caso migliore ed il caso peggiore dei due algoritmi.
2. Posto $n = length(A)$ si compili la tabella:

$$\begin{array}{lcllcl} C^{min}_{Ins}(n) & = & ??? & C^{min}_{Sel}(n) & = & ??? \\ C^{max}_{Ins}(n) & = & ??? & C^{max}_{Sel}(n) & = & ??? \\ S^{min}_{Ins}(n) & = & ??? & S^{min}_{Sel}(n) & = & ??? \\ S^{max}_{Ins}(n) & = & ??? & S^{max}_{Sel}(n) & = & ??? \end{array}$$

   Dove

$$\begin{array}{lcl} C^{min}(n) & = & \text{n. confronti nel caso migliore} \\ C^{max}(n) & = & \text{n. confronti nel caso peggiore} \\ S^{min}(n) & = & \text{n. spostamenti nel caso migliore} \\ S^{max}(n) & = & \text{n. spostamenti nel caso peggiore} \end{array}$$

In entrambi i punti si giustifichi la risposta.

Capitolo 4

Relazioni di ricorrenza

Esercizio 1. Risolvere la relazione di ricorrenza seguente calcolandone il Θ:

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

Esercizio 2. Si consideri la relazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2T(n-1) + n & \text{se } n > 0 \end{cases}$$

1. Si determini l'O-grande di T senza usare il master theorem.
2. Si scriva lo pseudo-codice di un algoritmo che calcoli $T(n)$ in tempo $\Omega(T(n))$.

Esercizio 3. Per risolvere una relazione di ricorrenza col *metodo del cambio della variabile* ci si basa sull'osservazione che, se $T(f(m)) = S(m) = O(g(m))$ ed f è invertibile, allora $T(n) = O(g(f^{-1}(n)))$. In tal modo la variabile $n = f^{-1}(m)$ viene sostituita con m per inferire l'ordine di grandezza di $T(n)$ quando sia noto, o più facile da trovare, l'ordine di grandezza di $S(m)$.

Si applichi dunque il metodo del cambio di variabile per stabilire l'O-grande di T che soddisfa la relazione:

$$T(n) = T(\sqrt{n}) + 1.$$

Esercizio 4. (Cormen 4.1-5) Dimostrare che la relazione di ricorrenza:

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$$

è $O(n \log n)$.

Suggerimento: si faccia un cambio di variabile usando la funzione $f(n) = 2n + 34$.

Esercizio 5. Si consideri la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} a & \text{se } n = 1 \\ 2T(\lceil n/2 \rceil) + b & \text{se } n > 1 \end{cases}$$

dove a e b sono due costanti intere.

- Si trovi $f(n)$ tale che $T(n) \in \Theta(f(n))$ senza usare l'apposito teorema.
- Si scriva in pseudo-codice un algoritmo ricorsivo $\textsc{Max}(A[1..n])$, dove $A[1..n]$ è un array di n interi. L'algoritmo deve restituire il valore massimo in A e la sua complessità temporale deve essere $T(n)$.

Capitolo 5

Analisi degli algoritmi

Esercizio 1. Siano n, r interi positivi ed x reale $\neq 0$, si considerino gli algoritmi:

```
POW(x, n)
  if n = 0 then return 1
  else
      y ← POW(x, ⌊n/2⌋)
      if n mod 2 = 0 then return y · y
      else return x · y · y

GAI(n, r)
  result ← 0
  for x ← 0 to n do
      result ← result + POW(x, r)
  return result
```

Si stabilisca il Θ di GAI(n, r) in termini di n, r, in modo dettagliato e senza far uso del master theorem.

Esercizio 2. Si consideri il seguente algoritmo, dove $V[0..n-1]$ è un vettore di n interi:

```
JACK(V[0..n − 1])
  m ← 0
  for i ← 0 to n − 1 do
      j ← i
      while j < n and (V[j] mod 2) = 1 do
          j ← j + 1
      m ← max(m, j − i)
  return m
```

Si risponda alle seguenti domande, giustificando le risposte.

1. Qual è il significato del numero m calcolato da JACK(V)?
2. Quale la sua complessità in n?
3. Sapreste fornire un algoritmo equivalente, ma di complessità asintoticamente inferiore?

Esercizio 3. Si consideri il seguente algoritmo.

```
P(A[1..n], l, r)
  if l = r then
```

```
    for i = 1 to n do
        stampa A[i]
else
    for i = l to r do
        scambia A[l] con A[i]
        P(A[1..n], l + 1, r)
        scambia A[i] con A[l]
```

- Cosa stampa la chiamata $\mathrm{P}(A[1..n], 1, n)$ se A contiene elementi distinti? Suggerimento: simulare l'esecuzione con un vettore di tre elementi.
- Ricavare una relazione di ricorrenza che caratterizza il tempo di esecuzione di P. Suggerimento: scrivere la relazione di ricorrenza con una funzione a due variabili, $T(n, k)$, dove $k = r - l + 1$.
- Dimostrare che $T(n, n) \in O(n \cdot n!)$.

Esercizio 4. Si analizzi la complessità temporale in funzione di n del seguente algoritmo, ricavando la relazione di ricorrenza e calcolandone il Θ:

```
BUMP(n)
  if n ≤ 1 then return 5
  else
      m ← 0
      for i ← 1 to 8 do
          m ← m + BUMP(⌊n/2⌋)
      for i ← 1 to n do
          for j ← 1 to n do
              m ← m + 1
      return m
```

Esercizio 5. Si considerino i seguenti algoritmi:

1. ```
   BART(n)        ▷ n intero positivo
       result ← 0
       for i ← 2 to n do
           result ← result + i · FOO(i)        ▷ FOO(i) richiede tempo Θ(log i)
       return result
   ```

2. ```
   HOMER(n)       ▷ n intero positivo
       if n = 1 then
           return 1
       else
           temp ←HOMER(⌈n/2⌉)+HOMER(⌊n/2⌋)
           for i ← 1 to n do
               temp ← temp + i
           return temp
   ```

3. ```
   MARGE(n)       ▷ n intero positivo
       if n = 1 then
           return 1
       else
           temp ←MARGE(⌈n/2⌉)+MARGE(⌊n/2⌋)
           for i ← 1 to n do
               for j ← 1 to n do
                   temp ← temp + i · j
   ```
   ```

return *temp*

Si stabilisca l'ordine di grandezza Θ del tempo dei tre algoritmi in funzione di n, e se ne giustifichi la risposta mostrando la costruzione con cui si è ottenuto il risultato.
Suggerimento. Nel caso di BART è utile considerare il limite:

$$\lim_{n\to\infty} \frac{\log(n!)}{n \log n} = 1$$

Negli altri due casi si può usare il master-theorem, specificando e calcolando il valore dei parametri α, β.

Esercizio 6.

Si consideri l'algoritmo:

```
MOO(A[1..n])        ▷ A array di interi
temp ← 0
for i ← 1 to n do
    for j ← i + 1 to n do
        if |A[i] − A[j]| > temp then
            temp ← |A[i] − A[j]|
return temp
```

Si richiede di:

1. spiegare brevemente che cosa calcola MOO, *senza* dire come lo calcoli;
2. stabilire l'ordine di grandezza Θ del tempo di questo algoritmo;
3. definire un secondo algoritmo GOO che ritorni lo stesso risultato, ma in tempo asintoticamente inferiore.

Esercizio 7. Si considerino i seguenti algoritmi:

```
TIM(A[i..j])        ▷ A array di interi, i ≤ j
if i = j then
    return A[i]
else
    m ← ⌊(i + j)/2⌋
    return TIM(A[i..m]) + TIM(A[m + 1..j])

KIM(A[i..j])        ▷ A array di interi, i ≤ j
if i = j then
    return A[i]
else
    m ← ⌊(i + j)/2⌋
    r ← KIM(A[i..m])
    for k ← m + 1 to j do
        r ← r + A[k]
    return r
```

Si risponda alle domande:

1. Gli algoritmi TIM e KIM sono equivalenti?
2. Quali sono le loro complessità?

Esercizio 8. Sia $A[0..n-1]$ un vettore di interi distinti, ed $n = 2k+1$ per qualche $k \geq 0$ (pre-condizione). Si consideri la procedura:

```
SAM(A)
  i ← −1, m ← 0, M ← n − 1
  while m ≠ M do
      i ← i + 1, m ← 0, M ← 0
      for j ← 0 to n − 1 do
          if A[j] < A[i] then m ← m + 1
          if A[j] > A[i] then M ← M + 1
  return A[i]
```

Si risponda alle seguenti domande, giustificando le risposte:

1. Se e quando SAM(A) termina, che relazione sussiste tra $A[i]$ ed il vettore $A[0..n-1]$?
2. Se $A[0..n-1]$ soddisfa la precondizione, SAM(A) termina?
3. Qual è il Θ della complessità di SAM in termini di n nel caso peggiore?
4. Esiste un algoritmo equivalente a SAM, ma asintoticamente più efficiente?

Capitolo 6

Liste e tabelle hash

6.1 Liste

Le liste sono rappresentate con puntatori a record di due campi *head* e *next*, contenenti l'informazione associata all'elemento di testa della lista ed il puntatore alla coda rispettivamente; la lista vuota è rappresentata dal puntatore *nil*. Negli svolgimenti si usino le funzioni HEAD(L) e TAIL(L) che, se $L \neq nil$, ritornano $L.head$ e $L.next$ rispettivamente. La funzione CONS(x, L) alloca un nuovo record N, assegna $N.head \leftarrow x$ e $N.next \leftarrow L$, quindi ritorna N.

Esercizio 1. Si dia lo pseudo-codice della funzione MULTIPLES(L) che data una lista semplice L di n elementi restituisca il numero degli elementi che hanno almeno una copia in L. Ad esempio: se $L = [3, 2, 3, 3, 5, 2]$ allora MULTIPLES(L) $= 2$.
Suggerimento. Una semplice soluzione è $O(n^2)$, ma ne esiste una $O(n \log n)$.

Esercizio 2. Sia il *rango* di un elemento in una lista di interi la somma di quell'elemento con tutti quelli che lo seguono. Si scriva un algoritmo ottimo RANK(L) che distruttivamente modifichi L in modo che gli elementi della lista risultante siano i ranghi degli elementi della lista data. Ad esempio: se $L = [3, 2, 5]$ allora diventa $L = [10, 7, 5]$.

Esercizio 3. Si dia lo pseudo-codice della funzione REVERSE(L) che data una lista semplice L di n elementi restituisca la lista inversa L^{-1}, costituita da tutti gli elementi di L in ordine inverso. Si richiede inoltre che REVERSE non sia distruttiva, ossia non alteri L, e che operi in tempo $O(n)$.

Esercizio 4. Una lista si dice *palindroma* se se le sue due metà, eliminando eventualmente l'elemento di posto medio se la lunghezza della lista è dispari, sono simmetriche ossia l'una l'inversa dell'altra.

Si dia lo pseudo-codice della funzione PALINDROME(L) che data una lista semplice L di n elementi decida se L è palindroma in tempo $O(n)$.

Esercizio 5. Si scriva un algoritmo ODDEVEN(L) ottimo che data una lista L di interi ne riordini distruttivamente gli elementi in modo che i dispari precedano i pari. L'algoritmo deve essere stabile, nel senso che l'ordine relativo tra i dispari e tra i pari deve essere preservato. Ad esempio: se $L = [3, 7, 8, 1, 4]$ allora si ottiene $L = [3, 7, 1, 8, 4]$.
Suggerimento. L'algoritmo ODDEVEN(L) restituisca una tripla $Odd, Last, Even$, dove Odd è la lista degli elementi dispari in L, $Even$ quella dei pari; essendo tuttavia L una lista semplice viene calcolato anche $Last$, che è l'ultimo elemento della lista dei dispari, il cui successivo deve essere $Even$. Si noti che, se in L non ci sono dispari $Odd = Last = nil$, cosa di cui si deve tener conto tanto quando si aggiunge un dispari in testa ad Odd, quanto quando si aggiunge un pari davanti ad $Even$. Una volta eseguito ODDEVEN(L) la lista richiesta sarà Odd se $Odd \neq nil$, $Even$ altrimenti.

Esercizio 6. Supponiamo di rappresentare insiemi finiti di interi con liste ordinate e senza ripetizioni. Si descrivano gli algoritmi INTERSECTION(A, B), UNION(A, B), DIFFERENCE(A, B) e SYMDIFFERENCE(A, B)

corrispondenti agli operatori insiemistici:

$$\begin{array}{rcll} A \cap B & = & \{x \mid x \in A \wedge x \in B\} & \text{intersezione} \\ A \cup B & = & \{x \mid x \in A \vee x \in B\} & \text{unione} \\ A \setminus B & = & \{x \mid x \in A \wedge x \notin B\} & \text{differenza} \\ A \Delta B & = & \{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\} & \text{differenza simmetrica} \end{array}$$

Tutti gli algoritmi devono essere ricorsivi e $O(n)$ dove n è la somma delle cardinalità, ossia delle lunghezze delle liste che rappresentano gli insiemi A, B. Inoltre, sebbene $A \Delta B = (A \cup B) \setminus (A \cap B)$ e la composizione di algoritmi lineari abbia costo lineare, si richiede di implementare SymDifference(A, B) direttamente, ad imitazione degli algoritmi per gli altri operatori.

Esercizio 7.

Si consideri il seguente algoritmo:

```
Foo(L)        ▷ L lista semplice di interi
M ← Cons(0, nil)
P ← M
while L ≠ nil do
    n ← 0
    Q ← L
    while Q ≠ nil do
        n ← n + Q.key
        Q ← Q.next
    P.next ← Cons(n, nil)
    P ← P.next
    L ← L.next
return M.next
```

Si richiede di:

1. spiegare brevemente che cosa ritorna Foo, *senza* dire come lo calcoli;
2. stabilire l'ordine di grandezza Θ del tempo di questo algoritmo;
3. definire un secondo algoritmo Goo che ritorni lo stesso risultato, ma in tempo asintoticamente inferiore.

6.2 Tabelle hash

Esercizio 8. Sia $T[0..m-1]$ una tabella hash con capacità $m = 10$ e chiavi intere, ed $h(k) = k \mod 10$ una funzione hash. Si disegni lo stato di T dopo aver inserito le chiavi $88, 12, 2, 22, 33$ utilizzando il metodo di indirizzamento aperto con ispezione lineare.

Esercizio 9.

- Si consideri una tabella hash ad indirizzamento aperto $T[0, ..., 10]$, con 11 elementi, che utilizza la seguente funzione di hashing

  $$h(k, i) = (k \ mod \ 11 + 2i + i^2) \ mod \ 11$$

  Si riporti il suo contenuto dopo l'inserimento delle seguenti chiavi: 18, 32, 7, 15.

- Si consideri una tabella di hash ad indirizzamento aperto $T[0, ..., 11]$, con 12 elementi, che utilizza la seguente funzione di hashing

  $$h(k, i) = (k \ mod \ 12 + 6i) \ mod \ 12$$

  Si riporti una sequenza di tre chiavi diverse tale che l'inserimento della terza chiave non sia possibile.

Capitolo 7

Alberi

7.1 Alberi binari e k-ari

Negli esercizi che seguono si assume che gli alberi binari siano realizzati con record di tre campi: *key* per la chiave (di solito un intero), *left* e *right* per i puntatori al sottoalbero sinistro e destro rispettivamente. Se invece gli alberi sono k-ari allora la realizzazione utilizza record a tre campi con il campo key come sopra, il campo *child* per il puntatore al primo dei sottoalberi, il campo *sibling* per il puntatore al fratello.

Esercizio 1. Sia dato un albero T (non vuoto), rappresentato con puntatori *child* e *sibling*, nel quale ogni nodo è etichettato con un numero intero. Si dia un algoritmo che restituisca *true* se l'etichetta di ogni nodo che ha almeno un figlio è uguale alla somma delle etichette dei figli, e *false* altrimenti.

Esercizio 2. Sia dato un albero T (non vuoto) con chiavi intere, rappresentato con puntatori *child* e *sibling*. Si dia un algoritmo ottimo SommaRamo(T) che aggiunga ad ogni foglia di T un nodo avente come chiave la somma di tutte chiavi dalla radice alla foglia.

Suggerimento: si definisca una funzione ausiliare ricorsiva SommaCammino(S, k) che dipende da un intero k in cui si accumula la somma delle chiavi incontrate sul cammino sino al nodo puntato da S.

Esercizio 3. Sia dato un albero T (non vuoto), rappresentato con puntatori *child* e *sibling*. Si dia un algoritmo ottimo di nome NumNodiProfondi(T, h) il quale, presi come argomenti il nodo radice T dell'albero e un intero h, restituisca il numero dei nodi dell'albero che si trovano a livello $\leq h$ (dove si assume come 0 il livello della radice). L'algoritmo non deve usare funzioni o procedure ausiliarie, né variabili esterne o campi aggiuntivi nei record che rappresentano i nodi.

Esercizio 4. Un albero binario non vuoto T è completo se per ogni $k \leq height(T)$ il livello k ha esattamente 2^k nodi. Si dia un algoritmo asintoticamente ottimo per decidere se T è completo.

Si dia una procedura ottima basata sulla BFS per decidere se T sia completo. Si consideri quindi la seguente definizione alternativa di albero binario completo: T è completo se costituito da un solo nodo oppure se ha due sottoalberi completi di uguale altezza. Si dimostri che le due definizioni sono equivalenti; quindi si dia un algoritmo ottimo per decidere se T è completo basato sulla seconda definizione e ricorsivo (quindi basato su una DFS).

7.2 Alberi binari di ricerca

Esercizio 5. Scrivere una procedura ottima che dato un albero binario di ricerca T non vuoto con chiavi intere e due interi $a \leq b$ produca la stampa delle chiavi di T nell'intervallo $a..b$ in modo che risulti ordinata crescente.

Esercizio 6. Sia T un albero binario di ricerca con chiavi intere, e siano $a < b$ due chiavi di T. Si dia lo pseudocodice di un algoritmo ottimo AntenatoComune(T, a, b) che trovi l'antenato comune più vicino a quelli che contengono le chiavi a e b (se a è antenato di b allora l'algoritmo deve restituire a, e viceversa). Quindi si risponda alle domande:

1. Qual è il confine inferiore alla complessità del problema? Qual è la complessità dell'algoritmo proposto?
2. Se T è un albero rosso-nero, qual è la complessità dell'algoritmo?

Esercizio 7. Si scriva lo pseudo-codice di un algoritmo che decida in tempo $O(n)$ se un'albero binario non vuoto con chiavi intere è di ricerca, dove n è la cardinalità dell'albero.

Suggerimento: Si definisca un algoritmo IsBinSearch(T) che ritorni una tripla b, m, M (o se si preferisce un record con questi tre campi) dove b è **true** o **false** a seconda che T sia di ricerca oppure no; se b è **true** allora m ed M sono il minimo ed il massimo delle chiavi in T rispettivamente.

7.3 Alberi ed ordinamento

Esercizio 8.

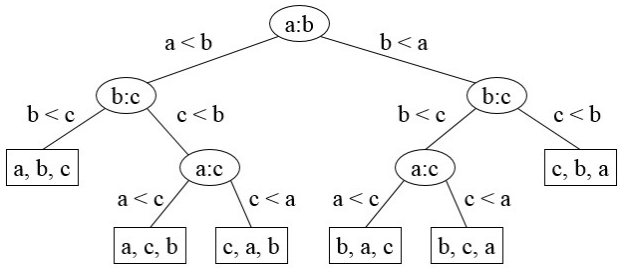
- Riportare le caratteristiche di un *heap minimo*.
- Il seguente array rappresenta un *heap minimo*.

$$(1, 2, 3, 10, 8, 9, 7, 14, 15, 16, 13)$$

  Effettuare l'estrazione del minimo disegnando lo heap dopo ogni scambio di elementi.

Esercizio 9. Si dimostri che, poiché ogni algoritmo basato sui confronti per ordinare n elementi richiede tempo $\Omega(n \log n)$, ogni algoritmo basato sui confronti per costruire un albero binario di ricerca di n elementi richiede ugualmente tempo $\Omega(n \log n)$.

Esercizio 10. L'albero seguente, detto albero delle decisioni, rappresenta i confronti necessari per ordinare un vettore di tre elementi a, b, e c.



Si disegni l'albero delle decisioni per un vettore di quattro elementi, a, b, c e d, assumendo $a > b$ (cioè bisogna disegnare solo metà dell'albero).

Capitolo 8

Grafi

8.1 Simulazione di algoritmi noti

Esercizio 1.

1. Si effettui la visita in ampiezza (BFS) del seguente grafo orientato a partire dal nodo a riportando l'albero generato dalla visita e il contenuto della coda utilizzata durante la visita dopo gli inserimenti degli adiacenti non ancora visitati di ciascun vertice:

$$\begin{aligned} a &: d, e \\ b &: a, c, e, f \\ c &: a, e \\ d &: b, c, e \\ e &: a, d \\ f &: d, e \end{aligned} \tag{8.1}$$

2. Ricordando che un grafo di dice *completo* se ogni coppia di vertici è collegata da un arco, si disegni l'albero generato da una visita in ampiezza effettuata su un grafo completo con $n + 1$ vertici.

Esercizio 2. Si applichi l'algoritmo di Dijkstra al grafo riportato sotto con le liste di adiacenti a partire dal nodo A. Riportare il contenuto della coda di priorità prima di ogni estrazione.

$$\begin{aligned} A &: B(3), C(6), D(7) \\ B &: C(2), E(1) \\ C &: D(1) \\ D &: \\ E &: D(4) \end{aligned}$$

Esercizio 3. Si applichi l'algoritmo di Dijkstra al grafo orientato pesato rappresentato con le liste di adiacenza qui sotto riportato (il numero tra parentesi è il peso dell'arco), a partire dal nodo A:

- A: B(3), C(2), D(7)
- B: E(4)
- C: B(5), D(4), E(6), F(6), G(9)
- D: E(7), G(3)

- E: F(4), G(4)
- F: G(2)
- G:

Si spieghi a parole l'uso della coda di priorità nell'esecuzione dell'algoritmo di Dijkstra.

8.2 Problemi

Esercizio 4. Un grafo non orientato $G = (V, E)$ si dice *bipartito* se esiste un insieme $S \subseteq V$ tale che per ogni $(u, v) \in E$ si abbia $u \in S$ e $v \in V \setminus S$ oppure $v \in S$ e $u \in V \setminus S$.

Si trovi un algoritmo di complessità $O(|V| + |E|)$ per decidere se G sia bipartito.

Esercizio 5. Si dimostri che il test di aciclicità per un grafo orientato $G = (V, E)$ può essere realizzato con un algoritmo di complessità $O(|V| + |E|)$.

Esercizio 6. Si dimostri che se G è un grafo non orientato connesso con archi pesati, allora esiste un MST di G, senza usare la correttezza di Kruskal o Prim.