Esercizi del corso di Algoritmi e strutture dati

LT di Informatica - Università di Torino

Andras Horvath, Ugo de'Liguoro

16 marzo 2021

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.

Soluzione 1.

- L'invariante del ciclo è
$$y = x^2 \wedge x \leq z.$$
- Prima di eseguire la riga 5 per la prima volta abbiamo $x = 0, y = 0$ da cui $0 = 0^2$ e $0 \leq z$ ossia la precondizione.

  Assumiamo che $x < z$ e che prima di eseguire la riga 6 abbiamo $y = x^2$. Denotiamo con x' e y' i nuovi valori di x e y dopo la riga 7. Abbiamo
$$x < z \Rightarrow x' = x + 1 \leq z.$$
  Inoltre
$$y = x^2 \Rightarrow y' = y + 2x + 1 = x^2 + 2x + 1 = (x + 1)^2 = x'^2$$
  Quindi il ciclo mantiene l'invariante.
- All'uscita dal ciclo $\neg(x < y)$ e $x \leq z$, quindi $x = z$ e $y = x^2 = z^2$.

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.

Soluzione 2. La versione ricorsiva si ottiene implementando direttamente le equazioni:

```
FastExp-Rec(x, n)
        ▷ Pre: x ≠ 0 reale, n ≥ 0 intero
        ▷ Post: ritorna il reale x^n
  if n = 0 then return 1
  else
      y ← FastExp-Rec(x, ⌊n/2⌋)        ▷ ip. ind. y = x^⌊n/2⌋
      if n mod 2 = 0 then               ▷ se k = ⌊n/2⌋ allora n = 2k
          return y · y
      else                              ▷ se k = ⌊n/2⌋ allora n = 2k + 1
          return y · y · x
```

Versione iterativa. Osserviamo che:

1. se $y = x$, $k = n$ e $z = 1$ si ha immediatamente che $y^k \cdot z = x^n \cdot 1 = x^n$;
2. se $k = 0$ e $x^n = y^k \cdot z$ allora $x^n = y^0 \cdot z = z$;
3. se $k = 2k' > 0$ e $x^n = y^k \cdot z$ allora: $x^n = y^{2k'} \cdot z = (y \cdot y)^{k'} \cdot z$;
4. se $k = 2k' + 1$ e $x^n = y^k \cdot z$ allora: $x^n = y^{2k'+1} \cdot z = (y \cdot y)^{k'} \cdot (y \cdot z)$.

Da questo si ottengono il seguente algoritmo e la sua prova di correttezza:

```
FastExp-It(x, n)
        ▷ Pre: x ≠ 0 reale, n ≥ 0 intero
        ▷ Post: ritorna il reale x^n
  y ← x, k ← n, z ← 1          ▷ y^k · z = x^n · 1 = x^n
  while k > 0 do               ▷ inv. x^n = y^k · z
      if k mod 2 = 1 then
          z ← z · y
      y ← y · y
      k ← ⌊k/2⌋
  return z
```

In conclusione si osservi che il numero delle moltiplicazioni effettuate da entrambe le versioni dell'algoritmo è proporzionale al numero di volte in cui n può essere diviso per 2, ossia a $\log_2 n$. Questo algoritmo è dunque certamente migliore di quello ingenuo che calcola x^n con $n - 1$ moltiplicazioni.

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

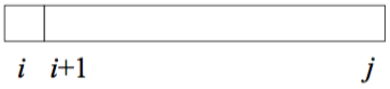
Ingresso: $A[i..j]$ un array di interi, un intero a.

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.

Soluzione 3. L'algoritmo ricorsivo si basa sulla suddivisione in tre casi:

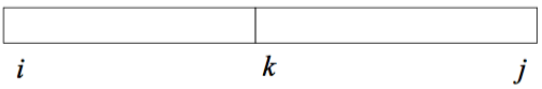
1. $i > j$: allora $A[i..j] = \emptyset$ onde $a \notin A[i..j]$ e l'algoritmo deve ritornare -1;
2. $i \leq j$ e $A[i] = a$: allora l'algoritmo deve ritornare i;
3. $i \leq j$ e $A[i] \neq a$: allora $a \in A[i..j]$ se e solo se $a \in A[i+1..j]$.




```
LinSearch-Rec(A[i..j], a)
      ▷ Pre: A[i..j] un array di interi, un intero a
      ▷ Post: ritorna k = min{k' ∈ i..j | A[k'] = a} se esiste, −1 altrimenti
  if i > j then        ▷ a ∉ A[i..j] = ∅
     return −1
  else      ▷ i ≤ j
     if A[i] = a then
        return i
     else      ▷ a ∈ A[i..j] se e solo se a ∈ A[i + 1..j]
        return LinSearch-Rec(A[i + 1..j], a)
           ▷ Ip.ind.: ritorna k = min{k' ∈ i + 1..j | A[k'] = a} se esiste, −1 altrimenti
```

L'algoritmo iterativo si basa sull'invariante, dove $k \in i..j+1$:

$$\forall h \in i..k-1.\ A[h] \neq a.$$




```
LinSearch-It(A[i..j], a)
      ▷ Pre: A[i..j] un array di interi, un intero a
      ▷ Post: ritorna k = min{k' ∈ i..j | A[k'] = a} se esiste, −1 altrimenti
  k ← i
  while k ≤ j and A[k] ≠ a do        ▷ inv. ∀h ∈ i..k − 1. A[h] ≠ a
     k ← k + 1
  if k ≤ j then        ▷ A[k] = a
     return k
  else      ▷ k = j + 1 > j, dunque ∀h ∈ i..j. A[h] ≠ a
     return −1
```

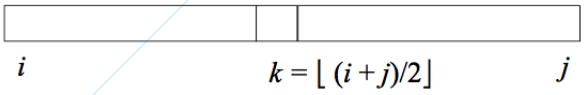
Esercizio 4. (**Ricerca dicotomica**) Si consideri il seguente problema:

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.

Soluzione 4. L'idea della ricerca dicotomica è di confrontare a con l'elemento di posto medio $k = \lfloor (i+j)/2 \rfloor$ e nel caso siano diversi restringere la ricerca all'intervallo $i..k-1$ se $a < A[k]$, $k+1..j$ se $A[k] < a$:




```
BinSearch-Rec(A[i..j], a)
      ▷ Pre: A[i..j] un array di interi ordinato, un intero a
      ▷ Post: ritorna un k ∈ i..j tale che A[k] = a se esiste, −1 altrimenti
  if i > j then        ▷ a ∉ A[i..j] = ∅
     return −1
  else      ▷ i ≤ j
     k ← ⌊(i + j)/2⌋        ▷ k punto medio di i..j
     if A[k] = a then return k
     else
        if a < A[k] then        ▷ ip. ind. (1.1)
           return BinSearch-Rec(A[i..k − 1], a)
        else      ▷ A[k] < a, ip. ind. (1.2)
           return BinSearch-Rec(A[k + 1..j], a)
```

Dimostriamo che la post-condizione vale per induzione (completa) su $n = |i..j| = j - i + 1$ se $i \leq j$, 0 altrimenti. Quando $i > j$ abbiamo che $n = 0$ e che $a \notin A[i..j] = \emptyset$ dunque la post-condizione è soddisfatta ritornando -1.

Altrimenti sia $i \leq j$ e $k = \lfloor (i+j)/2 \rfloor$ il punto medio di $i..j$, onde $|i..k-1| < n$ e $|k+1..j| < n$. Le ipotesi induttive sono:

$$\text{BinSearch-Rec}(A[i..k-1], a) = h \in i..k-1 \text{ t.c. } A[h] = a \text{ se esiste, } -1 \text{ altrimenti} \quad (1.1)$$

$$\text{BinSearch-Rec}(A[k+1..j], a) = h \in k+1..j \text{ t.c. } A[h] = a \text{ se esiste, } -1 \text{ altrimenti} \quad (1.2)$$

Se $A[k] = a$ allora k soddisfa la post-condizione. Se $a < A[k]$ allora, poiché $A[i..j]$ è ordinato, ogni $b \in A[k..j]$ è tale che $a < b$; quindi $a \in A[i..j]$ se e solo se $a \in A[i..k-1]$. Per l'ipotesi induttiva (1.1) abbiamo:

$$\text{BinSearch-Rec}(A[i..j], a) = \text{BinSearch-Rec}(A[i..k-1], a).$$

Similmente se $A[k] < a$ allora dall'ipotesi induttiva (1.2) concludiamo:

$$\text{BinSearch-Rec}(A[i..j], a) = \text{BinSearch-Rec}(A[k+1..j], a).$$

Versione iterativa:

```
BinSearch-It(A[i..j], a)
      ▷ Pre: A[i..j] un array di interi ordinato, un intero a
      ▷ Post: ritorna un k ∈ i..j tale che A[k] = a se esiste, −1 altrimenti
  p ← i      ▷ p confine inferiore dello spazio di ricerca
  q ← j      ▷ q confine superiore dello spazio di ricerca
  while p ≤ q do      ▷ inv. p..q ⊆ i..j ∧ ∀h ∈ i..j. A[h] = a ⇒ h ∈ p..q
     k ← ⌊(p + q)/2⌋      ▷ k punto medio di p..q
     if A[k] = a then return k
     else
        if a < A[k] then
           q ← k − 1
        else      ▷ A[k] < a
           p ← k + 1
  return −1
```

L'invariante è banalmente vero prima del **while** perché $p = i$ e $q = j$. Supponiamo che $p \leq q$, l'invariante valga prima dell'esecuzione del corpo e $k = \lfloor (p+q)/2 \rfloor$ sia il punto medio di questo intervallo. Tre casi sono possibili:

1. $A[k] = a$: allora l'esecuzione del corpo termina e $k \in p..q \subseteq i..j$ soddisfa la post-condizione.
2. $a < A[k]$: poiché $A[i..j]$ è ordinato e $p..q \subseteq i..j$, $A[p..q]$ è ordinato; quindi se $b \in A[k..q]$ allora $b \geq A[k] > a$; ne segue che se $a \in A[p..q]$ allora $a \in A[p..k-1]$, perciò l'invariante è soddisfatto ponendo $q = k - 1$.
3. $A[k] < a$: caso analogo al precedente, osservando che ora se $c \in A[p..k-1]$ allora $c < a$, onde se $a \in A[p..q]$ allora $a \in A[k+1..q]$ e l'invariante vale una volta posto $p = k + 1$.

Pertanto o l'esecuzione del corpo del **while** è interrotta, ed allora l'invariante implica che k soddisfi la post-condizione; oppure l'invariante è preservato e si torna ad eseguire il test di guardia. Se dunque il **while** termina senza mai eseguire **return** k, allora $p > q$ onde $p..q = \emptyset$. Per l'invariante, se vi fosse $h \in i..j$ tale che $A[h] = a$ allora h dovrebbe essere in $p..q$; per contrapposizione concludiamo che non esiste un $h \in i..j$ tale che $A[h] = a$; dunque correttamente l'algoritmo ritorna -1.

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.

Soluzione 5. Sia $n = length(A)$.

Invariante del **for** interno, linee 2-6:

$$A[j] = \min A[j..n]. \tag{1.3}$$

L'invariante è banalmente vero quando $j = n = length(A)$, il suo valore iniziale. Se $i + 1 \leq j < n$ e l'invariante vale in j, dobbiamo stabilire che sia conservato in $j - 1$, essendo il passo di questo **for** un decremento (espresso con **downto**).

Ora se $A[j-1] \leq A[j] = \min A[j..n]$ allora $A[j-1] = \min A[j-1..n]$. Se invece $b = A[j] < A[j-1] = a$ allora il test della linea 3 sarà vero e verrà eseguito lo scambio della linea 4, dopo il quale $A[j-1] = [b, a, A[j+1], \cdots, A[n]]$, onde $b < a = \min[a, A[j+1], \cdots, A[n]]$ e quindi $b = \min[b, a, A[j+1], \cdots, A[n]]$.

Invariante del **for** esterno, linee 1-7:

$A[1..i-1]$ è ordinato in senso non decrescente e
tutti gli el. di $A[i..n]$ sono $\geq$ di quelli in $A[1..i-1]$.

L'invariante è vero a vuoto quando $i = 1$, il suo valore iniziale, perché $A[1..0] = \emptyset$.

Supponiamo valga per i e dimostriamo che vale ancora per $i+1$. Quando il **for** interno termina $j = i$, onde il suo invariante (1.3) implica

$$A[i] = \min A[i..n] = A[i] \cup A[i+1..n]$$

onde $a = A[i] \leq p$ per ogni $p \in A[i+1..n]$. D'altronde a proviene per eventuali scambi dal semivettore $A[i..n]$ che contiene maggioranti degli elementi in $A[1..i-1]$ per l'ipotesi che l'invariante esterno valga fino ad i. Quindi $a \geq q$ per ogni $q \in A[1..i-1]$

$$A[1..i-1] \cup [a] \cup A[i+1..n]$$

è ordinato nella porzione $A[1..i]$ ed ha nella parte $A[i+1..n]$ solo maggioranti degli elementi in $A[1..i]$, come richiesto.

Per concludere che A' è ordinato basta osservare che all'uscita del **for** della linea 1 si ha $i = n$; l'invariante esterno ora implica che $A'[1..n-1]$ è ordinato in senso non decrescente e che $A'[n]$ maggiorizza tutti gli elementi di $A'[1..n-1]$, onde $A'[1..n]$ è ordinato.

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

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

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

1: $y \leftarrow 0$
2: **for** $i \leftarrow n$ **downto** 0 **do**
3: $\quad y \leftarrow a_i + x \cdot y$

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?

Soluzione 6. Sia $0 \leq i \leq n$. Assumiamo $y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$ prima dell'esecuzione del corpo dell'iterazione; poiché al termine dell'esecuzione del corpo i viene decrementato di 1, bisogna dimostrare:

$$a_i + x \cdot \left(\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \right) = \sum_{k=0}^{n-i} a_{k+i} x^k$$

Ora:

$$a_i + x \cdot \left(\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \right) = a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1} = a_i + \sum_{k=1}^{n-i} a_{k+i} x^k = \sum_{k=0}^{n-i} a_{k+i} x^k$$

osservando che $x \cdot x^{n-(i+1)} = x^{n-(i+1)+1} = x^{n-i}$.

L'invariante è banalmente verificato quando $i = n$:

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

dove si ricordi che la sommatoria vuota vale 0.

Al termine del **for** abbiamo $i = -1$, dunque sostituendo nell'invariante si ottiene:

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

come richiesto.

La soluzione ingenua per il calcolo di $P(x) = \sum_{k=0}^{n} a_k x^k$ è la seguente:

```
1: y ← 0
2: for i ← 0 to n do          ▷ inv. y = Σ_{k=0}^{i-1} a_k x^k
3:     z ← 1
4:     for j ← 1 to i do      ▷ inv. z = x^{j-1}
5:         z ← z · x
6:     y ← y + a_i · z
```

Questo algoritmo calcola $i - 1$ prodotti ad ogni esecuzione del **for** interno, linee 4-5, cui va aggiunto nel **for** esterno, linee 2-6, il prodotto alla linea 6. Quindi il numero di prodotti è $\sum_{i=0}^{n} i = n(n+1)/2$, proporzionale ad n^2.

Un miglioramento abbastanza drastico si basa sull'osservazione che il calcolo di x^i può essere accumulato su z invece di essere ricalcolato ad ogni iterazione:

```
1: y ← 0
2: z ← 1
3: for i ← 0 to n do          ▷ inv. y = Σ_{k=0}^{i-1} a_k x^k ∧ z = x^{i-1}
4:     y ← y + a_i · z
5:     z ← z · x
```

Con questa versione otteniamo un algoritmo che effettua $2n$ moltiplicazioni, che sono comunque di più delle n moltiplicazioni dell'algoritmo di Horner. D'altra parte, poiché è necessaria almeno una moltiplicazione per ogni coefficiente tra $a_1, \ldots, a_n$, l'algoritmo di Horner non è migliorabile.

Capitolo 2

Crescita asintotica delle funzioni

Esercizio 1. Si dimostri che:

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

Soluzione 1. Per stabilire (1) basta notare che

$$3^{n-2} = \frac{3^n}{3^2} = \frac{1}{9}3^n$$

Dato che $3 < \pi$ e dunque $3^n < \pi^n$ per ogni n, a fortiori abbiamo:

$$\frac{1}{9}3^n < 3^n < \pi^n.$$

Per (2), usando la regola di l'Hopital, studiamo il limite:

$$\lim_{n\to\infty} \frac{\log^2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log^2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{2(\log n)\cdot(1/n)\ln 2}{\sqrt{n}/2n}$$

Posto $c = 2\ln 2$ si ha

$$\frac{2(\log n)\cdot(1/n)\ln 2}{\sqrt{n}/2n} = \frac{c\log n}{n}\cdot\frac{2n}{\sqrt{n}} = 2c\frac{\log n}{\sqrt{n}}$$

Poiché $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$, se ne conclude che

$$\lim_{n\to\infty} 2c\frac{\log n}{\sqrt{n}} = 2c \lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0.$$

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 $n \geq 1$.

Soluzione 2. La definizione di Ω è:

$$T(n) \in \Omega(f(n)) \iff \exists c > 0, n_0, \forall n \geq n_0.\ cf(n) \leq T(n)$$

Le funzioni

$$f(n) = n, \quad g(n) = 2n$$

soddisfano i criteri.

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

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

Soluzione 3. E' sufficiente porre $f_1 = g_1 = g_2$ scegliendo f_2 tale che $g_2 = o(f_2)$ (ad esempio $f_1 = g_1 = g_2 = n$ mentre $f_2 = n^2$). Allora si ha che $f_1(n) \in O(g_1(n))$, ed anche se $f_2 \notin O(g_2)$, l'antecedente è vero. Tuttavia $f_1(n) + f_2(n) \notin O(g_1(n) + g_2(n)) = O(g_i)$ per $i = 1, 2$.

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

Soluzione 4.

- $2^{n+1} = 2 \cdot 2^n \in O(2^n)$ prendendo 2 come costante moltiplicativa;
- $2^{2n} = 2^{n+n} = 2^n \cdot 2^n \notin O(2^n)$ perché non è possibile trovare una costante c che limiti 2^n;
- $4^n = (2 \cdot 2)^n = 2^n \cdot 2^n \notin O(2^n)$ come nel caso precedente.

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.

Soluzione 5. La funzione $f(n) = n^2 = 2^{\log_2 n^2} = 2^{2 \log_2 n}$ appartiene a $2^{O(\log n)}$ prendendo $c = 2$. D'altra parte

$$\lim_{n \to \infty} \frac{n \log n}{n^2} = \lim_{n \to \infty} \frac{\log n}{n} = 0$$

onde $n \log n = o(n^2)$ ossia $n^2 \notin O(n \log n)$ ma $O(n \log n) \subset O(n^2) \subseteq 2^{O(\log n)}$, dove in realtà entrambe le inclusioni sono strette.

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

Soluzione 6. Base $h = 0$: allora $\sum_{i=1}^{n} i^0 = \sum_{i=1}^{n} 1 = n = O(n^{0+1})$. Passo $h > 0$: osserviamo che

$$\sum_{i=1}^{n} i^h = \sum_{i=1}^{n} i^{h-1} i \leq \sum_{i=1}^{n} i^{h-1} n = n \sum_{i=1}^{n} i^{h-1}$$

Per ipotesi induttiva $\sum_{i=1}^{n} i^{h-1} = O(n^h)$ onde $n \sum_{i=1}^{n} i^{h-1} = n\, O(n^h) = O(n^{h+1})$.

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

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

Soluzione 7. Per ipotesi per ogni $\varepsilon > 0$ esiste n_0 tale che per ogni $n > n_0$ abbiamo:

$$\left| \ell - \frac{f(n)}{g(n)} \right| < \varepsilon.$$

Da ciò segue che quasi ovunque:

$$\frac{f(n)}{g(n)} - 2\varepsilon < \ell \quad \implies \quad \frac{f(n)}{g(n)} < \ell + 2\varepsilon \quad \implies \quad f(n) < (\ell + 2\varepsilon) g(n),$$

ossia, essendo $\varepsilon, \ell > 0$ e dunque $\ell + 2\varepsilon > 0$, ne concludiamo che $f \in O(g)$. D'altra parte abbiamo anche che, quasi ovunque:

$$\frac{f(n)}{g(n)} + 2\varepsilon > \ell \quad \implies \quad \frac{f(n)}{g(n)} > \ell - 2\varepsilon \quad \implies \quad f(n) > (\ell - 2\varepsilon) g(n).$$

Scegliendo ε abbastanza piccola in modo che $\ell - 2\varepsilon > 0$, il che è sempre possibile poiché $\ell > 0$ (ad esempio $\varepsilon = \ell/4$) si conclude che $f \in \Omega(g)$ e quindi, combinando con quanto stabilito sopra, $f \in \Theta(g)$.

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:

$C^{min}_{Ins}(n)$	=	???	$C^{min}_{Sel}(n)$	=	???
$C^{max}_{Ins}(n)$	=	???	$C^{max}_{Sel}(n)$	=	???
$S^{min}_{Ins}(n)$	=	???	$S^{min}_{Sel}(n)$	=	???
$S^{max}_{Ins}(n)$	=	???	$S^{max}_{Sel}(n)$	=	???

Dove

$C^{min}(n)$	=	n. confronti nel caso migliore
$C^{max}(n)$	=	n. confronti nel caso peggiore
$S^{min}(n)$	=	n. spostamenti nel caso migliore
$S^{max}(n)$	=	n. spostamenti nel caso peggiore

In entrambi i punti si giustifichi la risposta.

Soluzione 1. Sia $n = length(A)$. Il caso migliore per INSERTION-SORT si verifica quando $A[1..n]$ sia già ordinato, perché in questo caso $A[i-1] \leq A[i]$ per ogni $i \in 2..n$. Ne segue che, essendo $j = i-1$ e $key = A[i]$ la prima volta che viene eseguito il test nella guardia del **while** interno alla linea 4, questo fallisce sempre e quindi viene eseguito $n-1$ volte, mentre il corpo non viene mai eseguito.

Il caso peggiore si verifica quando $A[1..n]$ sia ordinato in senso decrescente, perché allora per ogni $i \in 2..n$ e per ogni $j \in 1..i-1$ si ha che $A[j] > key = A[i]$. In questo caso il test nella linea 4 qui sotto viene eseguito $j+1$ volte per ogni esecuzione dell'intero blocco 4-6 ossia per j da 1 a $n-1$; allora la linea 4 viene eseguita $\sum_{j=1}^{n-1}(j+1)$ volte, mentre le linee 5, 6 vengono eseguite $\sum_{j=1}^{n-1} j$ volte ognuna.

La linea 1 viene eseguita sempre $n-1$ volte più un'ultima volta corrispondente all'uscita: n volte in tutto. Le linee 2, 3 e 7 vengono eseguite $n-1$ volte sia nel caso migliore che in quello peggiore.

	INSERTION-SORT	costo	min. n.	max n.
1.	**for** $i \leftarrow 2$ **to** $length(A)$	c_1	n	n
2.	$key \leftarrow A[i]$	c_2	$n-1$	$n-1$
3.	$j \leftarrow i-1$	c_3	$n-1$	$n-1$
4.	**while** $j > 0$ **and** $A[j] > key$	c_4	$n-1$	$\sum_{j=1}^{n-1}(j+1)$
5.	$A[j+1] \leftarrow A[j]$	c_5	0	$\sum_{j=1}^{n-1} j$
6.	$j \leftarrow j-1$	c_6	0	$\sum_{j=1}^{n-1} j$
7.	$A[j+1] \leftarrow key$	c_7	$n-1$	$n-1$

Da questa tabella ricaviamo le funzioni C_{ins} ponendo $c_4 = 1$ e $c_i = 0$ per $i \neq 4$:

$$C_{Ins}^{min}(n) = n-1$$

e

$$\begin{aligned} C_{Ins}^{max}(n) &= \sum_{j=1}^{n-1}(j+1) \\ &= n + \sum_{j=1}^{n-1} j \\ &= n + \frac{n(n-1)}{2} \\ &= n\,\frac{n+1}{2} \end{aligned}$$

La funzione $S_{Ins}^{min}(n)$ si ottiene ponendo $c_2 = c_7 = 1$ e $c_i = 0$ altrimenti:

$$S_{Ins}^{min}(n) = 2(n-1) = 2n-2$$

La funzione $S_{Ins}^{max}(n)$ si ottiene ponendo $c_2 = c_7 = c_5 = 1$ e $c_i = 0$ altrimenti:

$$\begin{aligned} S_{Ins}^{max}(n) &= 2(n-1) + \sum_{j=1}^{n-1} j \\ &= 2(n-1) + \frac{n(n-1)}{2} \\ &= (n-1)\left(2 + \frac{n}{2}\right) \\ &= (n-1)\,\frac{n+4}{2} \end{aligned}$$

Nel caso di SELECT-SORT, il caso migliore si ha quando A è ordinato e quello peggiore quando A è decrescente; tuttavia i tempi variano soltanto per il numero delle volte in cui viene eseguita la linea 6 qui sotto, dove non avvengono confronti né spostamenti. Quindi $C_{Sel}^{min}(n) = C_{Sel}^{max}(n)$ e $S_{Sel}^{min}(n) = S_{Sel}^{max}(n)$. Riportiamo la tabella di SELECT-SORT nel caso peggiore:

	SELECT-SORT	costo	max n.
1.	**for** $i \leftarrow 1$ **to** $length(A) - 1$	d_1	n
2.	$k \leftarrow i$	d_2	$n-1$
3.	**for** $j \leftarrow i+1$ **to** $length(A)$	d_3	$\sum_{j=1}^{n-1}(j+1)$
5.	**if** $A[k] > A[j]$ **then**	d_4	$\sum_{j=1}^{n-1} j$
6.	$k \leftarrow j$	d_5	$\sum_{j=1}^{n-1} j$
7.	scambia $A[i]$ con $A[k]$	d_6	$n-1$

Per calcolare $C_{Sel}(n)$ poniamo $d_4 = 1$ e $d_i = 0$ per $i \neq 4$, ottenendo:

$$C_{Sel}(n) = \sum_{j=1}^{n-1} j = n\,\frac{n-1}{2}$$

Per calcolare $S_{Sel}(n)$ poniamo $d_6 = 3$ dato che uno scambio comporta 3 spostamenti, e $d_i = 0$ per $i \neq 6$, ottenendo:

$$S_{Sel}(n) = 3(n-1)$$

La tabella dei confronti e degli spostamenti risulta:

$$\begin{aligned} C_{Ins}^{min}(n) &= n-1 & \qquad C_{Sel}^{min}(n) &= n(n-1)/2 \\ C_{Ins}^{max}(n) &= n(n+1)/2 & \qquad C_{Sel}^{max}(n) &= n(n-1)/2 \\ S_{Ins}^{min}(n) &= 2(n-1) & \qquad S_{Sel}^{min}(n) &= 3(n-1) \\ S_{Ins}^{max}(n) &= (n-1)(n+4)/2 & \qquad S_{Sel}^{max}(n) &= 3(n-1) \end{aligned}$$

In conclusione i casi peggiore e migliore per i due algoritmi coincidono; SELECT-SORT è migliore di INSERTION-SORT solo riguardo agli spostamenti nel caso peggiore, mentre INSERTION-SORT è nettamente superiore riguardo ai confronti nel caso migliore. Tuttavia i confronti nel caso peggiore e gli spostamenti in quello migliore sono quasi uguali.

Capitolo 4

Relazioni di ricorrenza

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

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

Soluzione 1. Svolgendo si ha:

$$\begin{aligned} T(n) &= T(n/2) + n \\ &= T(n/4) + n/2 + n \\ &= T(n/8) + n/4 + n/2 + n \\ &= T(n/2^k) + \textstyle\sum_{i=0}^{k-1} n/2^i \qquad \text{per } k \leq \log_2 n \end{aligned}$$

Scrivendo $\log n$ per $\log_2 n$, se $k = \log n$ allora $T(n/2^k) = 1$; d'altra parte, usando la formula per la progressione geometrica:

$$\sum_{i=0}^{\log n - 1} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i = n \cdot \frac{(1/2)^{\log n} - 1}{1/2 - 1}$$

da cui

$$n \cdot \left[\left(\frac{1}{2}\right)^{\log n} - 1\right] \cdot (-2) = -2n \cdot \left(\frac{1}{n} - 1\right) = 2n - 2 = \Theta(n)$$

In conclusione $T(n) = \Theta(n)$.

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 $\Omega(T(n))$.

Soluzione 2. Svolgendo la relazione otteniamo:

$$\begin{aligned} T(n) &= 2T(n-1) + n \\ &= 2(2T(n-2) + n - 1) + n = 4T(n-2) + 2n - 2 + n \\ &\leq 4T(n-2) + 2n + n \\ &\cdots \end{aligned}$$

da cui $T(n) \leq 2^k T(n-k) + n\sum_{i=0}^{k-1} 2^i$, per $0 \leq k \leq n$. Posto allora $k = n$ si ha:

$$T(n) \leq 2^n T(0) + n\sum_{i=0}^{n-1} 2^i = 2^n + n(2^n - 1) = O(n\,2^n).$$

Riguardo all'algoritmo per il calcolo di $T(n)$ una soluzione è:

```
T(n)
  if n = 0 then return 1
  else
      a ← T(n − 1)
      b ← T(n − 1)
      for i ← 0 to n − 1 do
          i ← i + 1
      return a + b + i
```

Si noti che il calcolo dei valori a e b, uguali a $T(n-1)$ e quindi uguali tra loro, viene ripetuto due volte per ottenere che la complessità dell'algoritmo sia $T(n)$ e quindi $\Omega(T(n))$, come richiesto.

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:

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

Soluzione 3. Osserviamo preliminarmente che (scrivendo $\log n$ per $\log_2 n$):

$$\sqrt{n} = \sqrt{2^{\log n}} = \left(2^{\log n}\right)^{\frac{1}{2}} = 2^{\frac{\log n}{2}}.$$

Posto allora $f(m) = 2^m$, con inversa $f^{-1}(n) = \log n$, abbiamo che

$$T(f(m)) = T(2^m) = T(\sqrt{2^m}) + 1 = T(2^{m/2}) + 1$$

Sia $S(m) = T(2^m)$; allora dall'equazione precedente deriva:

$$S(m) = S(m/2) + 1$$

per cui è nota la soluzione $S(m) = O(\log n)$. Applicando allora la sostituzione inversa otteniamo:

$$T(n) = O(\log f^{-1}(n)) = O(\log\log n).$$

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 $f(n) = 2n + 34$.

Soluzione 3. Definiamo la funzione:

$$f(n) = 2n + 34$$

da cui segue

$$f(n)/2 + 17 = n + 34 = f(n/2)$$

Allora si ha (trascurando gli arrotondamenti per difetto):

$$T(f(n)) = 2T(f(n)/2 + 17) + f(n) = 2T(f(n/2)) + f(n)$$

Posto $S(m) = T(f(m))$ la ricorrenza data diventa la ben nota equazione:

$$S(m) = 2S(m/2) + m = O(m \log m)$$

Visto che $n = f(n/2 - 17)$, abbiamo

$$T(n) = T(f(n/2 - 17)) = S(n/2 - 17)$$

da cui ricaviamo:

$$T(n) = O((n/2 - 17)\log(n/2 - 17)) = O(n \log n)$$

dove l'ultima equazione usa l'algebra di O-grande per la semplificazione.

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 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 $T(n)$.

Soluzione 4. Svolgendo l'equazione ed identificando $n/2$ con la sua parte intera superiore, si ottiene:

$$\begin{array}{rcl} T(n) & = & 2T(n/2) + b \\ & = & 2(2T(n/4) + b) + b = 4T(n/4) + 2b + b \\ & = & 4(2T(n/8) + b)) + 2b + b = 8T(n/8) + 4b + 2b + b \\ & \cdots & \end{array}$$

da cui

$$T(n) = 2^k T(n/2^k) + b\sum_{i=0}^{k-1} 2^i \qquad \text{dove } \; k \leq \log_2 n$$

Posto $k = \log_2 n$ si ha

$$T(n) = 2^{\log n} T(1) + (2^{\log n} - 1)b \approx na + nb = n(a + b) = \Theta(n).$$

Il seguente pseudo-codice descrive un algoritmo la cui complessità è quella richiesta:

```
MAX(A[i..j])
  if i > j then return −∞        ▷ A[i..j] = ∅ il cui max. è un valore < di ogni valore in A[1..n]
  else
    if i = j then return A[i]    ▷ A[i..j] = {A[i]}
    else
      m ← ⌊(i + j)/2⌋
      p ← MAX(A[i..m])
      q ← MAX(A[m + 1..j])
      return max(p, q)
```

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.

Soluzione 5. Il tempo dell'algoritmo POW(x, n) è limitato asintoticamnte dalla funzione $T(n) = T(n/2)+1$ (trascurando la parte decimale di $n/2$) per via della chiamata ricorsiva POW$(x, \lfloor n/2 \rfloor)$, che svolgendo risulta

$$T(n) = T(n/2^k) + k \quad \text{con} \quad k \leq \log_2 n$$

Sostituendo k con $\log_2 n$ si ottiene $T(n) = T(1) + \log_2 n = \Theta(\log n)$, dove $T(1)$ è una costante.

In GAI(n, r) il parametro n determina il numero di iterazioni del **for**, nel cui corpo le chiamate POW(x, r) hanno costantemente lo stesso valore del parametro r, che è quello che determina la complessità di POW. In conclusione la complessità di GAI(n, r) è $(n + 1) \cdot \Theta(\log r) = \Theta(n \log r)$.

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?

Soluzione 6. Il numero m che calcola JACK$(V[0..n-1])$ è il massimo numero di elementi dispari consecutivi in $V[0..n-1]$. JACK ha complessità quadratica nel caso peggiore che è quello in cui tutti gli elementi in $V[0..n-1]$ siano dispari. Infatti in questo caso il numero di volte che viene eseguito il ciclo più interno è

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{n} \in \Theta(n^2).$$

Esiste tuttavia un algoritmo equivalente di complessità $\Theta(n)$:

```
MAXODDSEQ(V[0..n − 1])
  m ← 0
  p ← 0
  for i ← 0 to n − 1 do
      if (V[i] mod 2) = 1 then        ▷ V[i] dispari
          p ← p + 1
      else      ▷ V[i] pari
          m ← max(m, p)
          p ← 0
  return max(m, p)
```

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 $T(n, n) \in O(n \cdot n!)$.

Soluzione 3. Simulando l'algoritmo sulla sequenza $1, 2, 3$ si ottiene la stampa delle sequenze:

$$1,2,3 \quad 1,3,2 \quad 2,1,3 \quad 2,3,1 \quad 3,2,1 \quad 3,1,2.$$

In generale $\mathrm{P}(A[1..n], 1, n)$ stampa tutte le permutazioni di $A[1..n]$. Infatti per ogni $1 \leq i \leq n$, l'algoritmo permuta $A[1]$ con $A[i]$, agisce su $A[2], \ldots, A[1], \ldots, A[n]$ ed effettua lo scambio inverso di $A[i]$ con $A[1]$. In questo modo porta in testa all'array tutti gli elementi $A[i]$ di $A[1..n]$, mentre la parte $A[2], \ldots, A[1], \ldots, A[n]$ possiamo assumere venga permutata per induzione sull'ampiezza dell'intervallo.

Che $T(n,n) \in O(n \cdot n!)$ si conclude argomentando che avviene una stampa di n elementi per ognuna delle $n!$ permutazioni di ordine n. Più precisamente sia $k = r - l + 1$. Poiché la stampa nel primo **for** richiede tempo n e viene eseguita quando $k = 1$, mentre il secondo **for**, che viene eseguito k volte quando $k > 1$, contiene due scambi di costo costante c ed una chiamata ricorsiva di costo $T(n, k-1)$, la relazione di ricorrenza risulta (trascurando la costante $2c$ perché ininfluente):

$$T(n,k) = \begin{cases} n & (k=1) \\ k \cdot T(n,k-1) & (k>1) \end{cases}$$

Infine, svolgendo questa ricorrenza a partire da $T(n,n)$, si ottiene:

$$\begin{aligned} T(n,n) &= n \cdot T(n,n-1) \\ &= n \cdot (n-1) \cdot T(n,n-2) \\ &= n \cdot (n-1) \cdot \ldots \cdot (n-k+1) \cdot T(n,k) \end{aligned}$$

da cui, posto $k = n-1$ si ha

$$n \cdot (n-1) \cdot \ldots \cdot 2 \cdot T(n,1) = n! \cdot n \in O(n \cdot n!).$$

Si osservi come, essendo k in $T(n,k) = k \cdot T(n,k-1)$ un parametro e non una costante, il main theorem non si applica.

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
```

Soluzione 7. Poiché il corpo dei **for** annidati viene eseguito n^2 volte, mentre BUMP($\lfloor n/2 \rfloor$) viene eseguito 8 volte, si ottiene la ricorrenza:

$$T(n) = 8T(n/2) + n^2.$$

Svolgendo si ottiene:

$$T(n) = 8T(n/2) + n^2 = 8(8T(n/2^2) + (n/2)^2) + n^2 = 8^2 T(n/2^2) + 2n^2 + n^2 = \cdots$$

da cui in generale, quando $k \leq \log_2 n$ (scritto semplicemente $\log n$ nel seguito):

$$T(n) = 8^k T(n/2^k) + \left(\sum_{i=0}^{k-1} 2^i \right) n^2 = 8^k T(n/2^k) + (2^k - 1)n^2.$$

Posto allora $k = \log n$ e supponendo $T(1) = 1$ abbiamo

$$T(n) = 8^{\log n} + (2^{\log n} - 1)n^2 = 2n^3 - n^2 = \Theta(n^3).$$

Esercizio 5. Si considerino i seguenti algoritmi:

1. BART(n) ▷ n intero positivo
   result ← 0
   for $i \leftarrow 2$ **to** n **do**

result $\leftarrow$ *result* $+ i \cdot$ Foo(i) $\triangleright$ Foo(i) richiede tempo $\Theta(\log i)$
return *result*

2. Homer(n) $\triangleright$ n intero positivo
if $n = 1$ **then**
return 1
else
temp $\leftarrow$Homer($\lceil n/2 \rceil$)+Homer($\lfloor n/2 \rfloor$)
for $i \leftarrow 1$ **to** n **do**
temp $\leftarrow$ *temp* $+ i$
return *temp*

3. Marge(n) $\triangleright$ n intero positivo
if $n = 1$ **then**
return 1
else
temp $\leftarrow$Marge($\lceil n/2 \rceil$)+Marge($\lfloor n/2 \rfloor$)
for $i \leftarrow 1$ **to** n **do**
for $j \leftarrow 1$ **to** n **do**
temp $\leftarrow$ *temp* $+ i \cdot j$
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.

Soluzione 8. Il tempo di Bart(n) è determinato dalla somma dei tempi di Foo(i) per $i = 2, \ldots, n$ che sappiamo essere $\Theta(\log i)$; quindi per qualche $c > 0$:

$$\begin{aligned} T_{Bart}(n) &= \textstyle\sum_{i=2}^{n} c \log i \\ &= c \textstyle\sum_{i=2}^{n} \log i = c\,(\log 2 + \log 3 + \ldots + \log n) \\ &= c \log(2 \cdot 3 \cdot \ldots \cdot n) = c \log(n!) \end{aligned}$$

Da $\lim_{n\to\infty} \frac{\log(n!)}{n \log n} = 1$ segue che $\log(n!) = \Theta(n \log n)$ da cui concludiamo che Bart(n) $= \Theta(n \log n)$.
Come detto nel suggerimento useremo il master-theorem per il quale in una ricorrenza della forma:

$$T(n) = aT(n/b) + cn^{\beta}$$

posto $\alpha = \log a / \log b$ si ha che

se $\alpha > \beta$ allora $T(n) = \Theta(n^{\alpha})$
se $\alpha = \beta$ allora $T(n) = \Theta(n^{\alpha} \log n)$
se $\alpha < \beta$ allora $T(n) = \Theta(n^{\beta})$

La relazione di ricorrenza dell'algoritmo Homer(n) risulta:

$$T_{Homer}(n) = 2T_{Homer}(n/2) + c\,n$$

Quindi $a = b = 2$ da cui $\alpha = 1 = \beta$ onde $T_{Homer}(n) = \Theta(n \log n)$.

A causa dei due **for** annidati, la relazione di ricorrenza di Marge(n) invece risulta:

$$T_{Marge}(n) = 2T_{Marge}(n/2) + c\,n^2$$

Come per Homer, $a = b = 2$ onde $\alpha = 1$, ma $\alpha = 1 < 2 = \beta$ onde $T_{Marge}(n) = \Theta(n^2)$.

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.

Soluzione 9. L'algoritmo Moo(A) calcola il massimo delle differenze degli elementi di $A[1..n]$ in valore assoluto; il tempo è determinato dai due cicli annidati, e precisamente:

$$\begin{aligned} T_{Moo}(n) &= (n-1) + (n-2) + \ldots + 2 + 1 \\ &= \textstyle\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2) \end{aligned}$$

Poiché il massimo delle differenze degli elementi di A è pari al valore assoluto della differenza tra il massimo ed il minimo in A possiamo definire Goo come segue:

```
Goo(A[1..n])        ▷ A array di interi
max ← min ← A[1]
for i ← 2 to n do
    if max < A[i] then
        max ← A[i]
    else
        if min > A[i] then
            min ← A[i]
return |max − min|
```

Visto che il corpo dell'unico **for** è $\Theta(1)$ e viene eseguito $n-1$ volte il tempo di Goo è $(n-1)\cdot\Theta(1) = \Theta(n)$.

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⌋
```

$r \leftarrow$ KIM$(A[i..m])$
for $k \leftarrow m+1$ **to** j **do**
  $r \leftarrow r + A[k]$
return r

Si risponda alle domande:

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

Soluzione 7. I due algoritmi calcolano la medesima funzione, e cioè la somma dei valori in $A[i..j]$:

$$\text{TIM}(A[i..j]) = \sum_{k=i}^{j} A[k] = \text{KIM}(A[i..j])$$

Nel caso di TIM questo segue per induzione completa sull'ampiezza $|i..j|$ dell'intervallo $i..j$. Infatti nel caso in cui $i = j$, si ha $|i..j| = 1$ e la tesi vale trivialmente; se $|i..j| > 1$ e $m = \lfloor (i+j)/2 \rfloor$ allora $|i..m| < |i..j|$ per ipotesi induttiva $\text{TIM}(A[i..m]) = \sum_{h=i}^{m} A[h]$; analogamente $\text{TIM}(A[m+1..j]) = \sum_{k=m+1}^{j} A[k]$, da cui essendo $i < j$ si ha che $\text{TIM}(A[i..j])$ è:

$$\text{TIM}(A[i..m]) + \text{TIM}(A[m+1..j]) = \sum_{h=i}^{m} A[h] + \sum_{k=m+1}^{j} A[k] = \sum_{k=i}^{j} A[k]$$

Nel caso di KIM$(A[i..j])$ si ragiona analogamente salvo che $\text{KIM}(A[i..m]) = \sum_{h=i}^{m} A[h]$ vale per induzione, mentre la somma $\sum_{k=m+1}^{j} A[k]$ è calcolata dal **for** successivo alla chiamata ricorsiva.

Anche le complessità coincidono. Infatti quella di TIM$(A[i..j])$ è limitata dalla relazione di ricorrenza:

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

Procedendo per sostituzione:

$$T(n) = 2T(n/2) + b = 2(2T(n/4) + b) + b = 4T(n/4) + 2b + b = 8T(n/8) + 4b + 2b + b = ...$$

e assumendo che la dimensione sia potenza di 2 e considerando che $n/2 + n/4 + \cdots + 4 + 2 + 1 = n - 1$

$$= n \cdot T(1) + n/2 \cdot b + n/4 \cdot b + \cdots + 4b + 2b + b = n \cdot a + (n-1)b$$

e quindi $T(n) \in \Theta(n)$.

La complessità temporale $S(n)$ di KIM con $n = j - i + 1$:

$$S(n) = \begin{cases} a & n = 1 \\ S(n/2) + b \cdot n & n > 1 \end{cases}$$

Procedendo per sostituzione:

$$S(n) = S(n/2) + b \cdot n = S(n/4) + b \cdot n/2 + b \cdot n = S(n/8) + b \cdot n/4 + b \cdot n/2 + b \cdot n = ...$$

e assumendo che la dimensione sia potenza di 2 e considerando che $2 + 4 + \cdots + n/4 + n/2 + n = 2n - 2$

$$= S(1) + b \cdot 2 + b \cdot 4 + \cdots + b \cdot n/4 + b \cdot n/2 + b \cdot n = a + b(2n - 2)$$

e quindi $S(n) \in \Theta(n)$.

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?

Soluzione 8. Il ciclo **for** calcola il numero m dei minoranti di $A[i]$ ed il numero M dei suoi maggioranti in $A[0..n-1]$; quindi se SAM(A) termina, per qualche i deve essere $m = M$ ovvero $A[i]$ ha lo stesso numero di minoranti e maggioranti in $A[0..n-1]$; in tal caso si dice che $A[i]$ è la *mediana* di $A[0..n-1]$.

Essendo $n = 2k+1$ dispari e gli elementi in $A[0..n-1]$ tutti distinti, esiste una permutazione ordinata di A tale che $A[i_0] < \cdots < A[i_k] < \cdots < A[i_{n-1}]$ dove $A[i_k]$ è la mediana; perciò SAM(A) termina se A soddisfa le pre-condizioni.

Dato che i assume valori compresi tra $0 = -1+1$ ed $n-1$ (per l'esistenza della mediana), il caso peggiore è quando $i_k = n-1$, quindi il **while** sarà eseguito al più n volte; essendo chiaro che il **for** interno ha complessità $\Theta(n)$, la complessità nel caso peggiore di SAM(A) è $n \cdot \Theta(n) = \Theta(n^2)$.

Infine dall'argomento sull'esistenza della mediana segue che per trovare $A[i_k]$ è sufficiente ordinare $A[0..n-1]$, che può essere fatto in tempo $O(n \log n)$.

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 $O(n \log n)$.

Soluzione 1. Una soluzione quadratica genera la lista M degli elementi che hanno un doppione in L e ne ritorna la lunghezza:

```
MULTIPLES(L)
  M ← nil
  while L ≠ nil do
      if MEMBER(HEAD(L), TAIL(L)) and not MEMBER(HEAD(L), M) then
          M ← CONS(HEAD(L), M)
      L ← TAIL(L)
  return LENGTH(M)
```

Sia n la lunghezza di L. La complessità di MEMBER(x, N) è lineare nella lunghezza della lista N; dato che le lunghezze di TAIL(L) e di M sono sempre limitate da n, il test dell'**if** è $O(n) + O(n) = O(n)$ ad ogni iterazione; ma il ciclo **while** viene eseguito n volte, dunque ha complessità $O(n^2)$. Infine LENGTH è $O(len(M)) = O(n)$; dunque MULTIPLES(L) è $O(n^2) + O(n) = O(n^2)$.

Una seconda soluzione, migliore anche se non asintoticamente, si basa sul conteggio delle occorrenze di un elemento in una lista. Supponiamo che COUNT(x, L) conti quante volte x occorre in L, cosa che si può fare in tempo $O(n)$ con una semplice scansione di L. Allora:

```
MULTIPLES'(L)
  if L = nil then
      return 0
  else
      if COUNT(HEAD(L), TAIL(L)) = 1 then
          return 1 + MULTIPLES'(TAIL(L))
      else
          return MULTIPLES'(TAIL(L))
```

La relazione di ricorrenza di MULTIPLES' è della forma $T(n) = T(n-1) + n = O(n^2)$.

Una soluzione $O(n \log n)$ ordina (una copia di) L e conta quanti segmenti di lunghezza > 1 contenga la lista ordinata. Si consideri infatti l'algoritmo:

```
MultiplesOrd(L)
      ▷ Pre: L ordinata
      ▷ Post: ritorna il numero degli elementi di L che abbiano almeno due occorrenze
  numS ← 0        ▷ conta i segmenti di elementi uguali di lunghezza > 1
  inS ← false     ▷ true se il cursore è in uno di questi segmenti
  while L ≠ nil and Tail(L) ≠ nil do
     if not inS and Head(L) = Head(Tail(L)) then
        numS ← numS + 1, inS ← true
     if Head(L) ≠ Head(Tail(L)) then
        inS ← false
     L ← Tail(L)
  return numS
```

Poiché MultiplesOrd scandisce L una volta facendo lavoro in tempo $O(1)$ ad ogni iterazione, ha tempo $O(n)$; quindi il tempo predominante sarà quello dell'ordinamento, che può farsi in tempo $O(n \log n)$.

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 $\textsc{Rank}(L)$ 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 $L = [10, 7, 5]$.

Soluzione 2. L'algoritmo $\textsc{Rank}(L)$ si basa sull'osservazione che se L e $\textsc{Tail}(L)$ non sono vuoti allora dopo la chiamata ricorsiva $\textsc{Rank}(\textsc{Tail}(L))$ la testa di $\textsc{Tail}(L)$ contiene la somma di tutti gli elementi successivi a $\textsc{Head}(L)$ nella lista data originariamente.

```
Rank(L)
  if L ≠ nil and Tail(L) ≠ nil then
     L.next ← Rank(Tail(L))
     L.head ← Head(L) + Head(Tail(L))
  return L
```

Un confine inferiore banale basato sulla dimensione dei dati è $\Omega(n)$; d'altra parte il tempo di $\textsc{Rank}(L)$ è $T(n) = T(n-1) + O(1) = O(n)$, dove n è la lunghezza di L. Quindi Rank è ottimo.

Esercizio 3. Si dia lo pseudo-codice della funzione $\textsc{Reverse}(L)$ 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 $O(n)$.

Soluzione 3. Supponiamo che $\textsc{Cons}(K, L)$ prenda in ingresso una chiave K ed una lista L e ritorni una lista con testa K e coda L, un algoritmo per il calcolo dell'inversa di L è il seguente:

```
Reverse(L)
  R ← nil
  while L ≠ nil do
     R ← Cons(L.key, R)
     L ← Tail(L)
  return R
```

La lista R è costruita come una pila, quindi inserendo in R gli elementi di L nell'ordine dato (cioè scandendo L in avanti) risulterà che l'ultimo elemento di L sarà il primo di R, il penultimo sarà il secondo e così via; dunque quando $\textsc{Reverse}(L)$ termina avremo che $R = L^{-1}$.

Dato poi che il corpo del **while** ha complessità $O(1)$ e che viene eseguito n volte, dove n è la lunghezza di L, $\textsc{Reverse}(L)$ opera in tempo $O(n)$ come richiesto.

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 $O(n)$.

Soluzione 4. L'algoritmo che segue si basa sull'osservazione che L è palindroma se uguale membro a membro alla sua inversa. Poiché $R \leftarrow$ REVERSE(L) dell'esercizio 3 calcola non distruttivamente l'inversa R di L operando in tempo $O(len(L)) = O(n)$, per risolvere il problema basta costruire una funzione EQUAL(L, R) che decida se L ed R siano uguali membro a membro ed operi in tempo $O(len(L) + len(R)) = O(n)$:

```
EQUAL(L, R)
  if L = nil then
      return R = nil
  else
      if R = nil then
          return L = nil
      else
          return HEAD(L) = HEAD(R) and EQUAL(TAIL(L), TAIL(R))
```

La relazione di ricorrenza di EQUAL è $T(n) = T(n-2) + O(1) = O(n)$, dove $n = len(L) + len(R)$. Allora l'algoritmo PALINDROME(L) risulta:

```
PALINDOME(L)
  R ← REVERSE(L)
  return EQUAL(L, R)
```

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.

Soluzione 5. Definiamo la funzione ausiliaria:

```
AUXODDEVEN(L)
  if L = nil then
      return nil, nil, nil
  else
      Odd, Last, Even ← AUXODDEVEN(TAIL(L))
      if Head(L) mod 2 = 1 then
          L.next ← Odd
          if Odd = nil then
              Last ← L, Last.next ← P
          Odd ← L
      else      ▷ Head(L) pari
          L.next ← Even
          if Last ≠ nil then
              Last.next ← L
          Even ← L
      return Odd, Last, Even
```

Allora l'algoritmo OddEven(L) può essere descritto come:

```
OddEven(L)
  Odd, Last, Even ← AuxOddEven(L)
  if Odd = nil then
     return Even
  else
     return Odd
```

Un confine inferiore banale basato sulla dimensione dei dati è $\Omega(n)$; d'altra parte il tempo dell'algoritmo AuxOddEven(L), che è quello di OddEven(L), è $T(n) = T(n-1) + O(1) = O(n)$, dove n è la lunghezza di L. Quindi OddEven è ottimo.

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:

$A \cap B$	$=$	$\{x \mid x \in A \wedge x \in B\}$	intersezione
$A \cup B$	$=$	$\{x \mid x \in A \vee x \in B\}$	unione
$A \setminus B$	$=$	$\{x \mid x \in A \wedge x \notin B\}$	differenza
$A \Delta B$	$=$	$\{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$	differenza simmetrica

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.

Soluzione 6. Le soluzioni qui di seguito proposte hanno tutte il medesimo impianto, simile alla fusione di liste ordinate, dove tuttavia occorre evitare, nell'intersezione e nell'unione, di produrre copie dello stesso elemento visto che l'uscita deve essere un insieme.

```
Intersection(A, B)
  if A = nil or B = nil then        ▷ ∅ ∩ B = A ∩ ∅ = ∅
     return nil
  else      ▷ A ≠ nil e B ≠ nil
     if Head(A) = Head(B) then
        return Cons(Head(A), Intersection(Tail(A), Tail(B)))
     else
        if Head(A) < Head(B) then       ▷ Head(A) ∉ B
           return Intersection(Tail(A), B)
        else      ▷ Head(A) > Head(B), quindi Head(B) ∉ A
           return Intersection(A, Tail(B))
```

```
Union(A, B)
  if A = nil then       ▷ ∅ ∪ B = B
     return B
  else
     if B = nil then        ▷ A ∪ ∅ = A
        return A
     else      ▷ A ≠ nil e B ≠ nil
        if Head(A) = Head(B) then
           return Cons(Head(A), Union(Tail(A), Tail(B)))
        else
           if Head(A) < Head(B) then        ▷ Head(A) ∉ B
              return Cons(Head(A), Union(Tail(A), B))
           else      ▷ Head(A) > Head(B), quindi Head(B) ∉ A
              return Cons(Head(B), Union(A, Tail(B)))
```

```
DIFFERENCE(A, B)
  if A = nil then                          ▷ ∅ \ B = ∅
    return nil
  else
    if B = nil then                        ▷ A \ ∅ = A
      return A
    else                                   ▷ A ≠ nil e B ≠ nil
      if HEAD(A) = HEAD(B) then
        return DIFFERENCE(TAIL(A), TAIL(B))
      else
        if HEAD(A) < HEAD(B) then          ▷ HEAD(A) ∉ B
          return CONS(HEAD(A), DIFFERENCE(TAIL(A), B))
        else                               ▷ HEAD(A) > HEAD(B), quindi HEAD(B) ∉ A
          return DIFFERENCE(A, TAIL(B))

SYMDIFFERENCE(A, B)
  if A = nil then                          ▷ ∅ΔB = B
    return B
  else
    if B = nil then                        ▷ AΔ∅ = A
      return A
    else                                   ▷ A ≠ nil e B ≠ nil
      if HEAD(A) = HEAD(B) then
        return SYMDIFFERENCE(TAIL(A), TAIL(B))
      else
        if HEAD(A) < HEAD(B) then          ▷ HEAD(A) ∉ B
          return CONS(HEAD(A), SYMDIFFERENCE(TAIL(A), B))
        else                               ▷ HEAD(A) > HEAD(B), quindi HEAD(B) ∉ A
          return CONS(HEAD(B), SYMDIFFERENCE(A, TAIL(B)))
```

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.

Soluzione 7. L'algoritmo FOO(L) genera una lista di interi, diciamo N, di uguale lunghezza rispetto ad L il cui i-esimo elemento è la somma di tutti gli elementi in L dal posto i in poi.

A causa dei due **while** annidati la complessità di FOO è quadratica in $n = length(L)$, e più precisamente, contando quante volte viene eseguito il corpo del **while** più annidato in cui viene scandita una porzione di L via via più corta di un elemento, risulta essere $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$.

L'algoritmo GOO si basa sull'osservazione che se si conooce $N[i+1]$, ossia l'elemento di posto $i+1$ in N, allora $N[i] = L[i] + N[i+1]$:

```
GOO(L)
  if L = nil then
      return nil
  else
      N ← GOO(TAIL(L))
      if N = nil then
          return CONS(HEAD(L), nil)
      else
          return CONS(HEAD(L) + HEAD(N), N)
```

La relazione di ricorrenza di GOO risulta:

$$T_{Goo}(n) = T_{Goo}(n-1) + \Theta(1)$$

da cui immediatamente si ottiene $T_{Goo}(n) = \Theta(n)$.

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.

Soluzione 8. Rappresentando T come un'array, al termine degli inserimenti si avrà (scrivendo *nil* per le locazioni libere):

$$\{nil, nil, 12, 2, 22, 33, nil, nil, 88, nil\}.$$

Le posizioni del 2 e del 22 si spiegano con il conflitto con 12; la posizione di 33 col fatto che $T[3] = 2$ al momento dell'inserimento, mentre la prima locazione libera ottenuta per scansione lineare è $T[5]$.

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.

Soluzione 9.

- La funzione di hashing deve essere considerata con i seguenti valori:

  $$h(18, 0) = 7$$
  $$h(32, 0) = 10$$
  $$h(7, 0) = 7, \ h(7, 1) = 10, \ h(7, 2) = 4$$
  $$h(15, 0) = 4, \ h(15, 1) = 7, \ h(15, 2) = 1$$

  La tabella dopo gli inserimenti è la seguente:

  $$(Nil, 15, Nil, Nil, 7, Nil, Nil, 18, Nil, Nil, 32, Nil)$$

- La sequenza 0,6,12 è tale perché

  $$h(0, 0) = 0$$
  $$h(6, 0) = 6$$
  $$h(12, 0) = 0, \ h(12, 1) = 6, \ h(12, 2) = 0, \ h(12, 3) = 6, ...$$

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.

Soluzione 1. Poiché T non è vuoro, $T \neq nil$; allora il problema è risolto dal seguente algoritmo in tempo $O(n)$ nella cardinalità di T:

```
FatherChildSum(T)
  if T.child = nil then return true
  else
      C ← T.child, s ← 0, b ← true
      while C ≠ nil do
          s ← s + C.key
          b ← b and FatherChildSum(C)
          C ← C.sibling
      return T.key = s and b
```

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.

Soluzione 2. Assumendo che S sia un albero non vuoto (quindi $S \neq nil$), la procedura SommaCammino(S, k) è definita:

```
SommaCammino(S, k)
  k ← k + S.key
  if S.child = nil then // S è una foglia
      S.child ← nuovo vertice V
      V.child ← V.sibling ← nil
      V.key ← k
  else
```

```
    S ← S.child
    while S ≠ nil do
        SommaCammino(S, k)
        S ← S.sibling
```

Quindi la procedura SommaRamo(T) consta della sola istruzione SommaCammino($T, 0$).

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.

Soluzione 3. Supposto che $h \geq 0$:

```
NumNodiProfondi(T, h)
  if T = nil then return 0
  else
    if h = 0 then return 1
    else      ▷  h > 0
      numNodi ← 1        ▷ per tener conto della radice
      T ← T.child
      while T ≠ nil do
        numNodi ← numNodi + NumNodiProfondi(T, h − 1)
        T ← T.sibling
      return numNodi
```

Si osservi che, anche se assumiamo che $T \neq nil$, nelle chiamate ricorsive prima poi accadrà che $T = nil$ a causa dell'assegnazione $T \leftarrow T.child$.

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).

Soluzione 4. Un confine inferiore basato sulla dimensione dei dati è $\Omega(n)$. Ora, se T è completo ed ha altezza h allora la sua cardinalità deve essere:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

Perciò un algoritmo ottimo è il seguente:

```
Complete(T)
  card ← Cardinality(T)
  height ← Height(T)
  return card = 2^(height+1) − 1
```

La complessità ottima di questo algoritmo presuppone che sia Cardinality che Height siano $O(n)$ nella cardinalità di T, il che è noto. Tuttavia l'uso dei due algoritmi comporta due visite dello stesso albero. Gli algoritmi che seguono risolvono lo stesso problema ciascuno con una singola visita di T.

Per contare quanti nodi appartengono a ciascun livello l'algoritmo BFS-Complete(T) gestisce una coda di coppie formate da un puntatore ad un sottoalbero di T ed un intero che ne memorizza il livello:

```
BFS-Complete(T)
        ▷ Pre: T albero binario non vuoto
        ▷ Post: true se T è completo
  level ← 0, card ← 1        ▷ card = cardinalità del livello level
  for all C figlio di T do
      Enqueue((C, 1), Q)
  while Q ≠ ∅ do
      (T', L) ← Dequeue(Q)
      for all C figlio di T' do
          Enqueue((C, L + 1), Q)
      if L = level then
          card ← card + 1
      else      ▷ L > level, dunque la visita del livello level è conclusa
          if card < 2^level then return false
          else
              level ← level + 1, card ← 1
  return card = 2^level
```

Poiché l'algoritmo BFS-Complete è una BFS ha complessità $O(n)$ nella cardinalità di T, ed essendo la visita consistente di un unico attraversamento dell'albero l'algritmo è migliore di quello basato sulla cardinalità e l'altezza.

Per dimostrare l'equivalenza tra le due definizioni di albero completo ragioniamo per induzione sull'altezza h di T.

$h = 0$ In questo caso T ha il solo livello 0 e $2^0 = 1$ ossia vi è un solo nodo; dunque le due definizioni coincidono.

$h > 0$ In questo caso, poiché $h \geq 1$, il livello 1 di T ha $2^1 = 2$ nodi, onde T ha due sottoalberi di altezza $h-1$. Ciascun livello k dei due sottoalberi, contiene la metà dei nodi del livello $k+1$ di T, che per ipotesi sono $2^{k+1} = 2 \cdot 2^k$, ossia il livello k dei due sottoalberi sinistro e destro di T contiene 2^k nodi. Per ipotesi induttiva i due sottoalberi sono completi nel senso della seconda definizione, ed avendo uguale altezza T stesso soddisfa la seconda definizione. Viceversa se T ha due sottoalberi di uguale altezza e T è alto h, le altezze dei due sottoalberi saranno $h-1$; per ipotesi d'induzione ciascun livello k dei due sottoalberi, contiene esattamente 2^k nodi, e come prima troviamo che, essendo il livello $k+1$ di T formato dai due livelli k dei suoi sottoalberi, tale livello avrà 2^{k+1} nodi come richiesto dalla prima definizione.

Ora, se si traducesse alla lettera la seconda definizione di completezza in un algoritmo di decisione avremmo una procedura di complessità $O(n^2)$ nel caso peggiore (sapreste dire quale sia il caso peggiore e perché la complessità di un algoritmo che calcolasse le altezze dei sottoalberi sarebbe quadratica?). Tuttavia il seguente algoritmo, che calcola l'altezza dei sottoalberi simultaneamente alla decisione sulla loro completezza, comporta un'unica visita, ed è dunque altrettanto efficiente di BFS-Complete.

```
DFS-Complete(T)
        ▷ Pre: T albero binario non vuoto
        ▷ Post: ritorna la coppia b, h dove se T è completo allora b = true ed h = height(T);
        ▷          b = false altrimenti
  if T.left = T.right = nil then return true, 0
  else
      if T.left = nil or T.right = nil then return false, _
      else
          bl, hl ← DFS-Complete(T.left)
          br, hr ← DFS-Complete(T.right)
          return (bl and br and hl = hr), hl + 1
```

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.

Soluzione 5. La procedura PRINTINT effettua una visita inordinata di T, limitandola alle chiavi comprese in $a \leq b$:

```
PRINTINT(T, a, b)
  if T.left ≠ nil and a < T.key then
      PRINTINT(T.left, a, b)
  if a < T.key < b then
      write T.key
  if T.right ≠ nil and b > T.key then
      PRINTINT(T.right, a, b)
```

Essendo una visita l'algoritmo è $O(n)$ nella cardinalità di T e dunque ottimo.

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?

Soluzione 6. L'antenato comune più vicino è il nodo tale che:

- contiene la chiave a (e la chiave b si trova nel suo sottoalbero destro),
- oppure contiene la chiave b (e la chiave a si trova nel suo sottoalbero sinistro),
- oppure la chiave a si trova nel suo sottoalbero sinistro e la chiave b si trova nel suo sottoalbero destro.

Quindi l'antenato comune si può trovare con un algoritmo che cerca una chiave x per la quale abbiamo $a \leq x \leq b$:

```
ANTENATOCOMUNE(T, a, b)
  if a ≤ T.key ≤ b then
      return T
  if b < T.key then
      return ANTENATOCOMUNE(T.left, a, b)
  if a > T.key then
      return ANTENATOCOMUNE(T.right, a, b)
```

La complessità è uguale a quella di una ricerca. In un albero binario di ricerca il confine inferiore alla complessità del problema è $\Omega(n)$ (dove n è il numero di nodi) perché nel caso l'albero sia degenere bisogna fare $n-1$ confronti per arrivare al nodo che contiene a o b. In un albero rosso-nero la complessità è $O(\log n)$ perché l'altezza dell'albero è tale.

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.

Soluzione 7.

```
IsBinSearch(T)
  if T.left = T.right = nil then return true, T.key, T.key
  else
      if T.right = nil then
          b, m, M ← IsBinSearch(T.left)
          return b ∧ M < T.key, m, T.key
      if T.left = nil then
          b, m, M ← IsBinSearch(T.right)
          return b ∧ m > T.key, T.key, M
           ▷ T ha sottoalberi sinistro e destro
      bL, mL, ML ← IsBinSearch(T.left)
      bR, mR, MR ← IsBinSearch(T.right)
      return (bL ∧ bR ∧ ML < T.key < mR), mL, MR
```

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.

Soluzione 8. Uno heap minimo è un albero binario semi-completo sinistro, vale a dire completo sino al penultimo livello mentre ciascun vertice dell'ultimo non ha lacune alla sua sinistra; inoltre la chiave di ogni vertice padre è minore o uguale di quella dei suoi eventuali figli. Nella rappresentazione in forma di array H l'eventuale figlio sinistro di $H[i]$ è $H[2i]$ ed il suo eventuale figlio destro è $H[2i+1]$.

Gli stati dello heap dato successivi all'estrazione del minimo in radice, ossia 1, durante il processo di ricostruzione dello heap sono:

$$(\underline{13}, \underline{2}, 3, 10, 8, 9, 7, 14, 15, 16)$$
$$(2, \underline{13}, 3, 10, \underline{8}, 9, 7, 14, 15, 16)$$
$$(2, 8, 3, 10, 13, 9, 7, 14, 15, 16)$$

Gli elementi sottolineati sono quelli coinvolti nello scambio che produce la riga successiva.

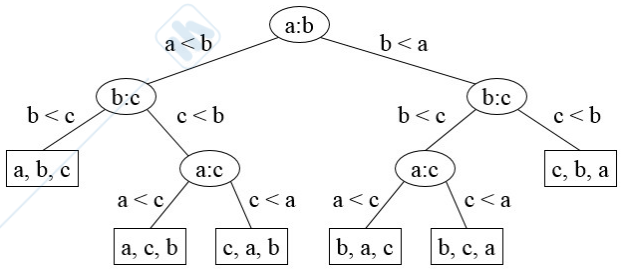
Esercizio 9. Si dimostri che, poiché ogni algoritmo basato sui confronti per ordinare n elementi richiede tempo $\Omega(n \log n)$, ogni algoritmo basato sui confronti per costruire un albero binario di ricerca di n elementi richiede ugualmente tempo $\Omega(n \log n)$.

Soluzione 9. Per assurdo sia BildRicTree un algoritmo in grado di costruire un albero di ricerca a partire da un array $A[1..n]$ di elementi in tempo $f(n) = o(n \log n)$. Si consideri allora l'algoritmo seguente:

```
TreeVisit(T, A, i)
  if T = ∅ then return i
  else
    i ← TreeVisit(T.left, A, i)
    A[i] ← T.key
    return TreeVisit(T.right, A, i + 1)
```

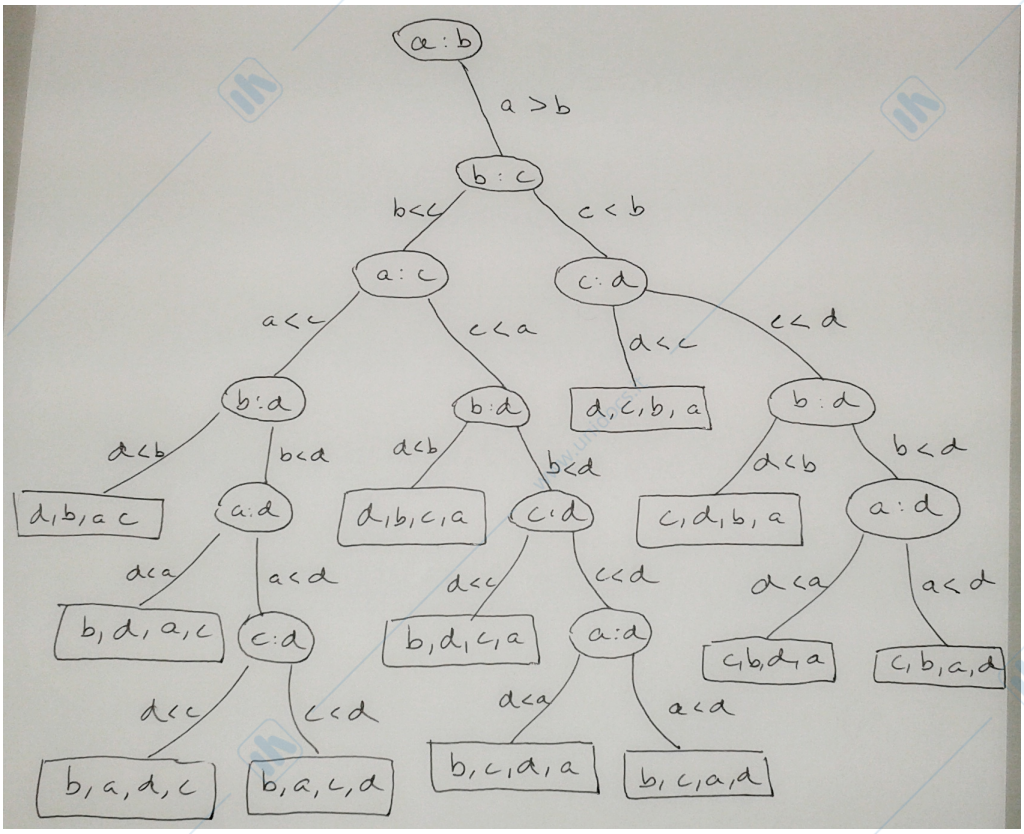
TreeVisit(T, A, i) copia in $A[i..n]$ le chiavi in T visitato inordinatamente (ritornando la prossima locazione libera in A ad uso delle chiamate ricorsive); dunque dopo l'esecuzione di TreeVisit(T, A, i) l'array $A[i..n]$ risulta ordinato. Visto che in TreeVisit$(T, A, 1)$ vi sono tante chiamate ricorsive quanti sono gli elementi di T, il tempo che richiede è $O(n)$. Se dunque prendiamo $T =$ BildRicTree$(A[1..n])$ e poi eseguiamo TreeVisit$(T, A, 1)$ otteniamo un algoritmo di ordinamento di A basato su confronti che opera in tempo $f(n) + n = o(n \log n)$, contraddicendo il teorema sul limite inferiore alla complessità degli algoritmi di ordinamento basati su confronti.

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).

Soluzione 10.



L'albero si può disegnare a partire da quello che riguarda tre elementi aggiungendo i confronti necessari per trovare la posizione del quarto elemento. L'albero non è unico, i confronti si possono eseguire in un ordine diverso che dà luogo ad un albero diverso.

Capitolo 8

Grafi

8.1 Simulazione di algoritmi noti

Esercizio 1.

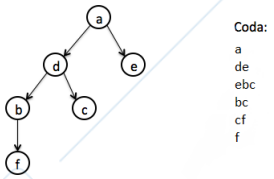
1. Si effettui la visita in ampiezza (BFS) del seguente grafo orientato a partire dal nodo a riportando l'albero generato dalla visita e il contenuto della coda utilizzata durante la visita dopo gli inserimenti degli adiacenti non ancora visitati di ciascun vertice:

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

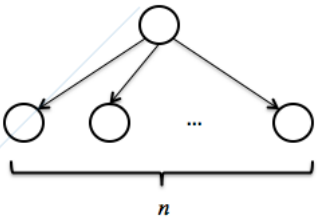
2. Ricordando che un grafo di dice *completo* se ogni coppia di vertici è collegata da un arco, si disegni l'albero generato da una visita in ampiezza effettuata su un grafo completo con $n+1$ vertici.

Soluzione 1.

1. L'albero generato dalla BFS del grafo a partire dal vertice a (a sinistra) e lo stato della coda dopo ciascun inserimento (a destra) risultano:



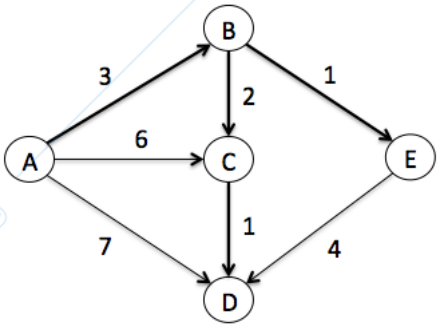
2. Poiché il grafo è completo, dato un vertice qualunque r esistono gli archi $(r, v_1), \ldots, (r, v_n)$ dove $v_1, \ldots, v_n$ sono i rimanenti n archi. Pertanto una BFS che cominci da un qualsiasi r genera un unico albero in cui tutti gli altri vertici (se ve ne sono, ossia se $n > 1$) sono figli del nodo r:



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.

$A : B(3), C(6), D(7)$
$B : C(2), E(1)$
$C : D(1)$
$D :$
$E : D(4)$

Soluzione 2. Il grafo, in cui gli archi di maggior spessore sone quelli che compongono i cammini minimi da A scelti dall'algoritmo di Dijkstra, risulta:



Quelli che seguono sono gli stati della coda (rappresentata come uno heap minimo in forma di array) prima dell'estrazione di ciascun vertice:

$(A, 0), (B, \infty), (D, \infty), (C, \infty), (E, \infty)$
$(B, 3), (D, 7), (C, 6), (E, \infty)$
$(E, 4), (D, 7), (C, 5)$
$(C, 5), (D, 7)$
$(D, 6)$

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.

Soluzione 3. La seguente tabella riporta il contenuto della coda di priorità prima di ogni estrazione (omettendo i nodi con distanza infinita e quelli già estratti).

B	C	D	E	F	G
3,A	**2,A**	7,A			
3,A		6,C	8,C	8,C	11,C
		6,C	7,B	8,C	11,C
			7,B	8,C	9,D
				8,C	9,D
					9,D

I cammini minimi con relativo peso sono

$$\begin{aligned} &A \to C : 2 \\ &A \to B : 3 \\ &A \to C \to D : 6 \\ &A \to B \to E : 7 \\ &A \to C \to F : 8 \\ &A \to C \to D \to G : 9 \end{aligned} \tag{8.2}$$

La coda di priorità contiene i nodi per i quali non si conosce il cammino minimo. La priorità associata a ciascun nodo è la stima del cammino minimo verso il nodo stesso. L'estrazione del nodo con stima minima corrisponde ad aver trovato il cammino minimo verso il nodo. Quando un nodo viene estratto, bisogna verificare se si può migliorare il cammino verso i nodi che sono ancora presenti nella coda.

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.

Soluzione 4. La soluzione proposta si basa su di una BFS modificata, iterata su tutti i vertici non ancora visitati, di cui ci limitiamo a mostrare l'algoritmo di visita da un vertice BIPARTITE-COMP, così chiamata perché esplora una delle componenti connesse di G a partire da un vertice nella componente. Assumiamo che il campo $v.color$ assuma uno tra i tre valori $bianco, rosso, noncolorato$ e che all'inizio tutti i vertici abbiano $v.color = noncolorato$:

```
BIPARTITE-COMP(G, r)        ▷ G = (V, E) non orientato, r ∈ V
  Q ← EMPTYQUEUE
  r.color ← bianco
  ENQUEUE(r, Q)
  bipartito ← true
  while Q ≠ ∅ do
      u ← DEQUEUE(Q)
      for all v ∈ Adj[u] do
          if v.color = noncolorato then
              if u.color = bianco then
```

```
            v.color = rosso
        else
            v.color = bianco
        EnQueue(v, Q)
    else
        if v.color = u.color then
            bipartito ← false
return bipartito
```

A rigore Bipartite-Comp(G, r) decide se la componente connessa cui appartiene r è 2-colorabile, ossia se i suoi vertici possono essere colorati in modo tale che i vertici di ogni arco $(u, v) \in E$ abbiano colore diverso.

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

Soluzione 5. Dimostriamo dapprima che G è ciclico se e solo se in ogni foresta generata da una visita DFS c'è almeno un arco all'indietro.

Se in una qualsiasi DF F di G c'è un arco all'indietro (v, u) allora esiste un albero $T \in F$ tale che v sia un discendente di u in T; perciò $u \rightsquigarrow v \to u$ è un ciclo.

Supponiamo che G sia ciclico. Siano F una DF di G e $C \subseteq V$ un ciclo di G tale che $C.d = \min\{u.d \mid u \in C\}$ sia minimo tra i tempi di scoperta in F di vertici in V appartenenti a qualche ciclo. Allora esistono $u, v \in C$ t.c. $u.d = C.d$ e $u \rightsquigarrow v$ e $(v, u) \in E$. Per la scelta di C abbiamo che $u.d < v.d$, dunque u, v appartengono allo stesso albero T che include C; ne segue che v è un discendente di u in T e (v, u) è un arco all'indietro di F.

Ora in una DFS la condizione per cui (v, u) è un arco all'indietro coincide con la scoperta di $u \in adj[v]$ prima che u esca dalla pila (la frontiera) ossia quando $u.color = grigio$; dunque è sufficiente modificare la DFS in modo che ritorni **false** non appena si scopre un arco all'indietro, e **true** se la visita di tutto il grafo termina senza scoprire un tale arco. La complessità dunque è quella della DFS ossia $O(|V| + |E|)$.

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.

Soluzione 6. Dimostriamo dapprima che esiste un albero T che sia una copertura di $G = (V, E)$. Se G è connesso ogni visita, non importa se BFS o DFS, genererà una foresta con un unico albero, visto che tutti i vertici sono raggiungibili gli uni dagli altri. Dunque quest'albero è una copertura di G. Poiché E è finito, anche l'insieme $\wp(E)$ delle parti di E è finito; ma un albero di copertura T (inteso come insieme di archi) è un sottoinsieme di E,dunque l'insieme degli alberi di copertura di G è un insieme $\{T_1, \ldots, T_n\} \subseteq \wp(E)$ finito non vuoto. Allora l'insieme di numeri reali $r = w(T)$ per qualche albero T di copertura di G è anch'esso finito e non vuoto; dunque ammette un minimo, corrispondente ad un MST di G.