Dispensa di Esercizi di Algoritmica (Parte I)

Esercizi su ordinamento, divide et impera, ricorrenze e heap (v1.1)

Chiara Baraglia, Andrea Lisi, Nicolas Manini, Davide Rucci

29 marzo 2019

Sommario

Questa dispensa contiene esercizi svolti utili alla preparazione dell'esame di Algoritmica. In particolare vi sono esercizi sulla prima parte del corso che comprende l'ordinamento, la tecnica divide et impera, le equazioni di ricorrenza e gli heap. Ogni esercizio è classificato per argomento in modo da guidare il lettore verso gli argomenti su cui vuole esercitarsi. Tutte le soluzioni sono dettagliatamente commentate per facilitarne la comprensione. Gli esercizi sono stati tratti principalmente da temi d'esame o compitini passati.

Alcuni esercizi hanno anche a supporto una implementazione in C dello pseudocodice. Tutti questi codici, insieme ad altri esempi, si trovano su GitHub.

Si prega di segnalare eventuali errori a `n.manini@studenti.unipi.it` e `d.rucci1@studenti.unipi.it`.

Esercizio 1 (Esame 22/06/2018). `Divide et Impera` `Relazioni di Ricorrenza`

Si progetti un algoritmo ricorsivo basato sulla tecnica Divide et Impera che calcola il secondo elemento più grande di un vettore A di n interi, senza modificarlo.

Si scriva la relazione di ricorrenza che descrive il suo costo in tempo al caso pessimo, e la si risolva.

Soluzione.

Partiamo da cosa dobbiamo fare: tramite un algoritmo ricorsivo, trovare il secondo elemento più grande dell'intero array. Essendo l'algoritmo Divide et Impera, andremo a trovare le soluzioni parziali all'interno di sotto-array, per poi combinarle ed ottenere la soluzione finale.

La nostra procedura divide l'array in due parti uguali ad ogni chiamata ricorsiva. Ogni chiamata ricorsiva risolverà il sotto problema resituendo la coppia <elemento maggiore, secondo elemento maggiore>.

```
Secondo(A, i, j) {
   if(j < i) return <−∞, −∞>
   if(j == i) return <A[i], −∞>
   m = (i+j)/2
   <p1, s1> = Secondo(A, i, m)
   <p2, s2> = Secondo(A, m+1, j)
   p = max(p1, p2)
   if(p == p1) s = max(s1, p2)
   else s = max(s2, p1)
   return <p, s>
}
```

Abbiamo 2 casi base: *i)* gli indici dei nostri array coincidono; *ii)* gli indici del nostro array si sorpassano (array vuoto). Nel caso *(i)* restituiamo una coppia: <elemento alla posizione degli indici, -infinito> (-infinito

è il secondo elemento più grande di un array di una posizione); nel caso *(ii)* restituiamo <-infinito, -infinito>. Nel caso generale in cui abbiamo le soluzioni parziali <p1, s1> e <p2, s2> cerchiamo i due elementi più grandi tra questi quattro valori: ci prendiamo il massimo tra p1, p2 e poi ci calcoliamo il massimo tra il p* scartato e s* del p* preso (che sarà la nostra soluzione).
Spiegazione: sappiamo che p1 > s1 e p2 > s2. Se p1 > p2 allora il secondo massimo sarà p2 o s1 (s2 non può essere perché p2 > s2, mentre non abbiamo ancora una relazione tra p2 e s1). Ragionamento equivalente se p2 > p1.
A questo punto restituiamo la coppia <p, s> che corrispondono al massimo ed al secondo massimo della porzione di array appena controllata. Implementazione in C dell'esercizio: Secondo.c.

Equazione di ricorrenza:

$$T(n) = \begin{cases} 2 * T(\frac{n}{2}) + \mathcal{O}(1), & \text{se } n \geq 2 \\ \mathcal{O}(1), & \text{se } n < 1 \end{cases}$$

Perché nel caso ricorsivo dividiamo l'array in due per due volte, mentre il codice per calcolare i massimi ed il codice all'interno dei casi base hanno un numero costante di istruzioni.
Per il Master Theorem, abbiamo $a = 2, b = 2$ e $f(n) = \mathcal{O}(1)$. Dunque $n^{\log_b a} = n^1$ e $f(n) = \mathcal{O}(n^{1-\epsilon})$ per un qualsiasi $0 \leq \epsilon < 1$. Ci troviamo quindi nel primo caso del teorema e possiamo concludere che $T(n) = \Theta(n)$. □

Esercizio 2 (Esame 22/06/2018). 🏷 `Relazioni di Ricorrenza`
Si consideri la seguente procedura ricorsiva, definita in funzione di un parametro (globale) intero $a > 0$:

```
foo(S, n) {
   if(n <= 1)
      return 5;
   InsertionSort(S, n);
   for i = 1 to a
      foo(S, n/2);
}
```

1. Scrivere la relazione di ricorrenza che descrive la complessità in tempo al caso pessimo di `foo(S,n)` in funzione di a;
2. Risolvere la relazione di ricorrenza al variare di a;
3. Stabilire per quali valori di a la procedura ha complessità in tempo al caso pessimo $\mathcal{O}(n^3)$.

Soluzione.
La relazione di ricorrenza sarà:

$$T(n) = \begin{cases} a * T(\frac{n}{2}) + \mathcal{O}(n^2), & \text{se } n > 1 \\ \mathcal{O}(1), & \text{se } n \leq 1 \end{cases}$$

poiché nel caso ricorsivo chiamiamo `foo` su metà degli elementi considerati nell'esecuzione in cui avviene al chiamata, e InsertionSort ha complessità $\mathcal{O}(n^2)$. Il caso base prevede che la funzione venga chiamata con un solo elemento e in questo caso termini restituendo "5", perciò abbiamo complessità $\mathcal{O}(1)$.
Osserviamo che $(n)^{\log_2(a)} = n^2 \leftrightarrow \log_2(a) = 2 \leftrightarrow a = 4$

I casi, pertanto, sono 3:

1. **Caso 1:** $n^2 = \mathcal{O}(n^{log_2(a)-\epsilon}) \leftrightarrow a > 4$
2. **Caso 2:** $n^2 = \Theta(n^{log_2(a)}) \leftrightarrow a = 4$
3. **Caso 3:** $n^2 = \Omega(n^{log_2(a)+\epsilon}) \leftrightarrow a < 4$.
   In questo caso dobbiamo considerare anche la condizione di regolarità: $a * f(\frac{n}{b}) \leq c * f(n)$.
   Abbiamo quindi $a * (\frac{n}{2})^2 \leq c * n^2 \leftrightarrow a * \frac{n^2}{4} \leq c * n^2$, pertanto basta prendere $\frac{a}{4} \leq c < 1$, che è possibile perché siamo nel caso in cui $a < 4$.

La soluzione è quindi ottenuta per casi:

1. $a > 4 \rightarrow \Theta(n^{log_2(a)})$;
2. $a = 4 \rightarrow \Theta(n^2 * log_2(n))$;
3. $a < 4 \rightarrow \Theta(n^2)$.

Per $a \leq 8$ si ha che $T(n) = \mathcal{O}(n^3)$. □

Esercizio 3 (Esame 13/07/2018). 🏷 `Heap`
Si fornisca l'algoritmo che inserisce una chiave in uno Heap di massimo e si discuta la sua complessità.

Soluzione.

```
Max_Heap_Insert(A, key) {
   A.heapsize++;
   A[A.heapsize] = key
   i = A.heapsize;
   while(i > 1 && A[i] > A[parent(i)]) {
      Scambia(A[i], A[parent(i)]);
      i = parent(i)
   }
}
```

$T(n) = \mathcal{O}(h) = \mathcal{O}(\log n)$, dove h è l'alteza dello heap. Questo perché ogni volta risaliamo l'albero tramite i puntatori al padre, quindi facciamo un numero di passi pari, al massimo, all'altezza dello heap che è logaritmica nel numero dei nodi. □

Esercizio 4 (Compitino 11/04/2017 - I). 🏷 `Divide et Impera` 🏷 `Relazioni di Ricorrenza`
Progettare un algoritmo di tipo Divide et Impera che abbia complessità in tempo:

- $T(n) = 3T(n/2) + \mathcal{O}(n \log n)$ per $n > 10$
- $\Theta(1)$ altrimenti.

Risolvere la ricorrenza associata.

Soluzione.

```
foo(S, n) {
   if(n <= 10)
      return 42;

   MergeSort(S, n);

   for i = 1 to 3
      foo(S, n/2);
}
```

Abbiamo che $a = 3$, $b = 2$ e $f(n) = n\log(n)$. Osserviamo che $\log_2(3) \approx 1.58$, e che quindi $f(n)$ risulta crescere in modo polinomiale superlineare in n. Segue quindi che ci troviamo nel caso in cui $f(n) = \mathcal{O}(n^{1.58-\epsilon})$ fissato ad esempio $\epsilon = 0.1$ (in generale per ogni $\epsilon \in (0, \log_2(3) - 1)$. Alla luce di ciò possiamo applicare il primo caso del Master Theorem e concludere che $T(n) = \Theta\left(n^{\log_2(3)}\right) \approx \Theta\left(n^{1.58}\right)$. □

Esercizio 5 (Compitino 11/04/2017 - II). 🏷 `Divide et Impera`
Dato un array ordinato $A[1 \dots n]$ contenente n elementi distinti nell'intervallo $[0, n + 1]$, si progetti un algoritmo efficiente che stabilisca se esiste $i \in [1, n]$ tale che $A[i] = i$. Si valuti la complessità in tempo dell'algoritmo proposto.

Soluzione.

```
find_i(A, sx, dx) {
   if(sx > dx) {
      return -1; // Non esiste
   }
   cx = ⌊(sx + dx)/2⌋;
   if(A[cx] == cx)
      return cx;
   if(A[cx] < cx)
      return find_i(A, cx + 1, dx); // Destra
   else
      return find_i(A, sx, cx); // Sinistra
}
```

L'algoritmo si basa sulla ricerca binaria del valore di cx all'interno dell'array, la differenza sta nella strategia con cui ricorriamo in uno dei due sottoarray. Il verificarsi della condizione $A[cx] < cx$ implica che il numero che stiamo cercando sia nella parte destra dell'array, infatti la parte sinistra conterrà tutti i numeri da 0 a $cx - 1$. Per questi motivi, $T(n) = \mathcal{O}(\log n)$. □

Esercizio 6 (Compitino 11/04/2017 - III). 🏷 `Heap`
Sia dato un vettore di interi $A = [6, 14, 2, 9, 7, 10]$, applicare l'algoritmo BUILD_MAX_HEAP(A) per la costruzione di uno heap di massimo, disegnando la configurazione del vettore dopo ogni iterazione del ciclo `for`.

Soluzione.
Termine della prima iterazione (i = 3):

$$[6, 14, 10, 9, 7, 2]$$

Termine della seconda iterazione (i = 2):

$$[6, 14, 10, 9, 7, 2]$$

Termine della terza iterazione (i = 1):

$$[14, 9, 10, 6, 7, 2]$$

Un'implementazione di un heap di massimo in C si può trovare qui: Heapsort.c. □

Esercizio 7 (Esame 11/04/2017). `Relazioni di Ricorrenza`
Risolvere la seguente ricorrenza al variare del parametro $a > 0$:

- $T(n) = aT(n/2) + \mathcal{O}(n^2)$ se $n > 0$
- $T(n) = \Theta(1)$ altrimenti.

Soluzione.
Abbiamo che i parametri del Master Theorem per questa ricorrenza sono $a = a$, $b = 2$ e $f(n) = n^2$, dobbiamo quindi studiare il comportamento di n^2 rispetto a $n^{\log_2(a)}$ al variare di a:

- Se $\log_2(a) = 2$, ossia per $a = 2^2$ abbiamo che $f(n) = \Theta(n^2)$ e quindi possiamo applicare il secondo caso del Master Th. concludendo che $T(n) = n^{\log_2 4} \log(n) = n^2 \log(n)$.
- Se invece $log_2(a) > 2$ abbiamo che esisterà sempre un valore $\epsilon > 0$ tale per cui $f(n) = \mathcal{O}(n^{\log_2(a)-\epsilon})$; poiché vale la disuguaglianza in senso stretto, infatti, sia $k = \log_2(a) - 2$, avremo che $k > 0$ e che la disuguaglianza iniziale è soddisfatta per ogni $\epsilon \in (0, k)$. In queste ipotesi possiamo quindi applicare il primo caso del Master Th. e concludere che $T(n) = \Theta(n^{\log_2(a)}) = \omega(n^2)$.
- Se infine $\log_2(a) < 2$, in modo duale a quanto sopra concludiamo che esisterà sempre un valore $\epsilon > 0$ tale per cui $f(n) = \Omega(n^{\log_2(a)+\epsilon})$; in particolare possiamo scegliere un qualsiasi $\epsilon \in (0, 2-\log_2(a))$. Per poter applicare il terzo caso del Master Th. dobbiamo tuttavia verificare che sia soddisfatta la condizione di regolarità i.e. che esista un valore $c < 1$ che soddisfi $a\left(\frac{n}{2}\right)^2 \leq cn^2$, ossia per cui $\frac{a}{4}n^2 \leq cn^2$ (per valori di n sufficientemente grandi). Possiamo quindi porre $c = \frac{a}{4}$ e la condizione risulta verificata come uguaglianza (più in generale potremmo prendere un qualunque $c \in \left[\frac{a}{4}, 1\right)$). Si noti che affinché l'ipotesi per cui $\log_2(a) < 2$ sia verificata abbiamo che a è strettamente minore di 4, ciò ci garantisce che il valore c così indicato soddisfi i vincoli imposti dalla condizione di regolarità ($\frac{a}{4} = c < 1$; nel caso più generale ciò ci garantisce che l'intervallo $\left[\frac{a}{4}, 1\right)$ contiene almeno un elemento, e che quindi è sempre possibile scegliere un valore per c); alla luce delle considerazioni fatte possiamo quindi applicare il terzo caso del Master Th. e concludere che $T(n) = \Theta(n^2)$.

□

Esercizio 8 (Compitino 14/04/2015 - I). `Progettazione di Algoritmi` `Limiti Inferiori`
Sia A un array di n interi distinti. Si consideri il problema di trovare la coppia $(A[i], A[j])$, con $i \neq j$, che **minimizza** la differenza in valore assoluto tra coppie di elementi, ovvero di trovare la coppia $(A[i], A[j])$ tale che:

$$|A[i] - A[j]| \leq |A[k] - a[l]|$$

$$\forall\ 1 \leq k, l \leq n$$

$$k \neq l.$$

1. Progettare un algoritmo efficiente che risolva il problema e discuterne la complessità;

2. Trovare un limite inferiore per il problema con la tecnica dell'albero di decisione e discuterne la significatività.

Soluzione — Punto 1.

```
CercaCoppia(A) {
   HeapSort(A);
   diff = |A[2] − A[1]|;
   coppia = ⟨A[2], A[1]⟩
   for i = 3 to n {
      if(|A[i] − A[i−1]| < diff) {
         diff = |A[i] − A[i−1]|;
         coppia = ⟨A[i], A[i−1]⟩;
      }
   }
   return coppia;
}
```

La complessità di questo algoritmo è $T(n) = T(heapsort) = T(n \log n)$, in quanto il sorting è l'operazione più costosa che viene eseguita, mentre il ciclo `for` esegue solo una scansione lineare dell'array.

□

Soluzione — Punto 2.

La tecnica dell'albero di decisione fornisce un limite inferiore pari a $\Omega(\log \#Soluzioni)$. Poiché stiamo cercando una coppia, il numero di soluzioni è dato dal numero di possibili coppie estratte da n elementi, che sono $\Theta(n^2)$. Si deriva quindi che la tecnica dell'albero di decisione fornisce $\Omega(\log n^2) = \Omega(\log n)$ come limite inferiore. Questo limite è poco significativo, in quanto dobbiamo scandire l'intero array almeno una volta perché uno qualsiasi dei suoi elementi potrebbe essere parte della soluzione. Un limite inferiore più significativo è quindi $\Omega(n)$. □

Esercizio 9 (Compitino 14/04/2015 - II). `Relazioni di Ricorrenza`

Si considerino le seguenti due equazioni di ricorrenza, dove a è un parametro:

$$T(n) = 8T(n/2) + \Theta(n) \tag{1}$$

$$T'(n) = aT'(n/4) + \Theta(n) \tag{2}$$

Risolvere le due equazioni. In particolare, si risolva la seconda equazione per i seguenti casi: $a < 4$, $a = 4$, $4 < a < 64$, $a = 64$, $a > 64$. Qual è il più piccolo valore di a per cui $T'(n)$ è asintoticamente **superiore** a $T(n)$?

Soluzione.

Prima di tutto risolviamo la ricorrenza per $T(n)$. Applicando il Master Theorem con parametri $a = 8, b = 2, f(n) = \Theta(n)$, abbiamo che $n^{\log_b a} = n^{\log_2 8} = n^3$ e quindi $f(n) = \mathcal{O}(n^{3-\epsilon})$ per ogni $\epsilon \in (0, 2)$. Dal primo caso del teorema otteniamo che $T(n) = \Theta(n^3)$.

Per la seconda ricorrenza distinguiamo i casi:

- $a < 4$: abbiamo che $\log_4 a < 1$ per cui $f(n) = \Omega(n^{(\log_4 a)+\epsilon})$ per $\epsilon \in (0, 1 - \log_4 a)$ e dobbiamo verificare la condizione di regolarità $a\frac{n}{4} \leq cn$ per $c < 1$. La condizione è soddisfatta per ogni $c \in [\frac{a}{4}, 1)$; tale intervallo è non vuoto poiché per ipotesi $a < 4 \rightarrow \frac{a}{4} < 1$. Segue quindi dal terzo caso che $T'(n) = \Theta(n)$.

- $a = 4$: abbiamo che $\log_4 a = 1$ per cui $f(n) = \Theta(n^{\log_4 a})$; segue quindi dal secondo caso che $T'(n) = \Theta(n \log n)$.
- $4 < a < 64$: abbiamo che $1 < \log_4 a < 3$ per cui $f(n) = \mathcal{O}(n^{(log_4 a)-\epsilon})$ per $\epsilon \in (0, (\log_4 a) - 1)$; segue quindi dal primo caso che $T'(n) = \Theta(n^{log_4 a}) = o(n^3)$.
- $a = 64$: abbiamo che $\log_4 a = 3$, per cui ci troviamo nella stessa situazione del punto precedente e possiamo concludere che $T'(n) = \Theta(n^3)$.
- $a > 64$: abbiamo che $\log_4 a > 3$ e come per il punto precedente otteniamo $T'(n) = \Theta(n^{\log_4 a}) = \omega(n^3)$.

Alla luce di questi risultati possiamo concludere che il più piccolo valore di a per cui $T'(n)$ è asintoticamente superiore a $T(n) = \Theta(n^3)$ è $a = 65$. Si noti che a è un numero intero in quanto indica il numero di sottoproblemi in cui viene suddiviso il problema iniziale. □

Esercizio 10 (Compitino 14/04/2015 - III). 🏷 `Progettazione di Algoritmi`
In riferimento all'esercizio 8, si consideri adesso il problema di trovare la coppia $(A[i], A[j])$, $i \neq j$ che **massimizza** la differenza in valore assoluto tra coppie di elementi. Descrivere un algoritmo che richiede tempo $\mathcal{O}(n)$. Questo algoritmo è ottimo?

Soluzione.
```
CercaCoppia(A) {
   min = minimo(A);
   max = massimo(A);

   return ⟨min, max⟩;
}
```

La soluzione è immediata in quanto, per un qualsiasi insieme di interi, la coppia che massimizza la differenza in valore assoluto è sempre quella costituita dal minimo e il massimo di esso. Poiché le ricerche del minimo e del massimo richiedono tempo $\Theta(n)$ concludiamo che $T(n) = \Theta(n)$. Si osservi che le due ricerche possono essere effettuate con un unico ciclo `for`.
L'algoritmo così individuato è ottimo in quanto la ricerca della coppia richiede di considerare almeno una volta tutti gli elementi in input. □

Esercizio 11 (Compitino 31/03/2016). 🏷 `Divide et Impera`
È dato un array ordinato $A[1 \dots n]$ contenente n interi distinti nell'intervallo $[1, n+1]$. Si vuole determinare l'intero mancante.

1. Descrivere un algoritmo di complessità $\mathcal{O}(n)$ e fornirne l'analisi al caso ottimo, medio e pessimo;
2. Descrivere un algoritmo Divide et Impera di complessità $\Theta(\log n)$;
3. Descrivere un algoritmo di complessità $\Theta(\log k)$, dove k è l'intero mancante.

Soluzione — Punto 1.
```
FindInt(A) {
   for i = 1 to n
      if(A[i] != i)
         return i;
   return n+1;
```

}

Poiché l'array è ordinato, abbiamo che $A[i] = i$ per tutti gli elementi che precedono l'intero mancante. Non appena questa condizione diventa falsa possiamo concludere che l'intero mancante sia i. Se invece arriviamo in fondo all'array possiamo concludere che l'intero mancante sia $n + 1$.
Il caso ottimo per questo algoritmo si verifica quando l'intero mancante è 1. Infatti in questo caso ci fermiamo alla prima iterazione del ciclo `for`, per cui $T(n) = \Theta(1)$ al caso ottimo.
Il caso pessimo si ha invece quando l'intero mancante è $n+1$, per cui eseguiamo tutte le $\Theta(n)$ iterazioni del `for` e solamente dopo possiamo rispondere con certezza che l'intero mancante è $n + 1$. Da questo si evince che $T(n) = \Theta(n)$ al caso pessimo.
Per il caso medio, assumendo che ogni intero abbia pari probabilità di non essere presente nell'array, possiamo immaginare che l'algoritmo termini dopo circa $n/2$ iterazioni. In notazione asintotica questo si traduce comunque in $T(n) = \mathcal{O}(n)$. □

Soluzione — Punto 2.
Per ottenere una complessità logaritmica nel numero di elementi dell'array, possiamo utilizzare una versione modificata della ricerca binaria. Osserviamo, infatti, che se l'intervallo fosse stato $[1, n]$ avremmo avuto $\forall\ i = 1, \ldots, n\ A[i] = i$. Dobbiamo quindi cercare il primo indice i che non soddisfi questa proprietà (ossia $A[i] \neq i$ e $A[i-1] = i-1$). Ad ogni chiamata ricorsiva quindi controlleremo se questa proprietà è soddisfatta dall'elemento intermedio cx e in caso di esito negativo ricorreremo in un sottoarray. In particolare l'intero mancante x induce uno shift di tutti i valori $j > x$ nell'array, per cui varrà $A[j] \neq j$. Concludiamo che se $A[cx] \neq cx$ (e cx non è soluzione, cioè $A[cx-1] \neq cx-1$) dobbiamo ricorrere nel sottoarray sinistro, altrimenti nel sottoarray destro.

```
FindIntBin(A, sx, dx) {
   if(dx < sx)
      return dx+1; // Notare che per questo specifico problema dx + 1 = n +1

   cx = ⌊(sx + dx) / 2⌋;
   if(cx == 1 && A[1] == 2)
      return 1;

   if(A[cx] != cx)
      if(A[cx-1] == cx-1)
         return cx;
      else
         FindIntBin(A, sx, cx-1);
   else
      FindIntBin(A, cx+1, dx);
}
```

Si osservi che vengono trattati separatamente i due casi particolari in cui l'intero mancante è 1 oppure $n + 1$: possiamo accorgerci del primo caso quando, al fine di verificare $A[cx - 1]$, andremmo ad accedere al valore $A[0]$, mentre restituiremo $n+1$ nel caso in cui la ricorsione venga effettuata sull'array vuoto in seguito ad una serie di sole chiamate ricorsive nelle parti destre.
La complessità di questo algoritmo è la stessa della ricerca binaria classica, ovvero $T(n) = \mathcal{O}(\log n)$ in quanto i controlli descritti sopra richiedono tempo $\Theta(1)$, il verificare se cx è soluzione richiede anch'esso tempo $\Theta(1)$

e ricorriamo solamente in una delle due metà in cui suddividiamo l'array.

□

Soluzione — Punto 3.

```
Doubling(A) {
   i = 1;
   while(2^i ≤ n && A[2^i] == 2^i)
      i++;
   if(2^i ≤ n)
      return FindIntBin(A, 2^(i-1), 2^i);
   else
      return FindIntBin(A, 2^(i-1), n);
}
```

Questo algoritmo utilizza la strategia dei "doubling jump", ovvero effettua dei "salti" nell'array lunghi potenze di 2 fino a quando non trova una posizione $j = 2^i$ per cui $A[j] \neq j$ oppure esce dall'array. Quando una di queste due condizioni si verifica abbiamo individuato la porzione di array $A[2^{i-1} \dots 2^i]$ in cui cercare l'intero mancante con la ricerca binaria definita al punto 2. La complessità di questo algoritmo sarà quindi proporzionale alla lunghezza della porzione di array in cui eseguiamo la ricerca. Dobbiamo dunque analizzare la lunghezza di questa porzione.

Ricordando che `FindIntBin` ha una complessità $\Theta(\log(dx - sx))$, osserviamo che questo algoritmo verrà invocato su una porzione lunga $\mathcal{O}(2^{i-1})$ (potrebbe essere più corta se $2^i > dx$). Questo deriva dal fatto che $2^i - 2^{i-1} = 2^{i-1}$. Intuitivamente, chiamato k l'intero mancante, dimostriamo che $k = \mathcal{O}(2^{i-1})$. Abbiamo $2^{i-1} < k \leq 2^i$ in quanto $k \in A[2^{i-1} \dots 2^i]$. Supponiamo per assurdo che la lunghezza 2^{i-1} dell'intervallo sia $> k$, allora 2^i non sarebbe la prima potenza di 2 dopo k in quanto $2^{i-1} > k$. Ciò è assurdo perché l'algoritmo individua la potenza 2^i affinché $2^{i-1} < k \leq 2^i$. Ne consegue che la lunghezza dell'intervallo in cui effettuiamo la ricerca binaria è $\mathcal{O}(k)$. Si noti infine che le iterazioni del `while` sono $\log k$ in quanto la condizione $2^i > k$ è verificata dopo $i = \log_2 k$ iterazioni. La complessità dell'algoritmo è quindi $T(n) = \Theta(\log k) + \mathcal{O}(\log k) = \Theta(\log k)$ dove il primo termine è dato dai doubling jump e il secondo dalla ricerca binaria. A livello di prestazioni abbiamo che $\Theta(\log k) = \mathcal{O}(\log n)$ in quanto $k \leq n + 1$. □

Esercizio 12 (Compitino 31/03/2016 - II). 🏷 `Relazioni di Ricorrenza`

Determinare il tempo di esecuzione al caso di pessimo di un `MergeSort` in cui una delle due chiamate ricorsive è sostituita da una chiamata a `QuickSort`.

Soluzione.

La nuova relazione di ricorrenza sarà $T(n) = T(n/2) + \mathcal{O}(n^2)$, in quanto viene eseguita soltanto una chiamata ricorsiva di MergeSort, mentre il lavoro che viene fatto al di fuori delle chiamate ricorsive diventa $\mathcal{O}(n^2)$, cioè il tempo di QuickSort. Si noti che il tempo necessario per l'operazione di Merge viene dominato dal tempo di esecuzione di QuickSort. Inoltre QuickSort, per quanto ricorsivo, non è una chiamata ricorsiva propria di questo algoritmo.

Per applicare il Master Theorem, abbiamo che $a = 1$, $b = 2$ e $f(n) = n^2$. Poiché $n^{\log_b a} = n^0$, si ha che $f(n) = n^2 = \Omega(n^{(\log_b a)+\epsilon})$, per un qualsiasi $0 < \epsilon \leq 2$. Ci troviamo quindi nel terzo caso del teorema e dobbiamo verificare che $af(n/b) \leq cf(n)$ per qualche $c < 1$ (*condizione di regolarità*). Sostituendo, abbiamo che $\frac{n^2}{4} \leq cn^2$, da cui possiamo ricavare $c \geq \frac{1}{4}$, per cui la condizione di regolarità è verificata per $\frac{1}{4} \leq c < 1$. Possiamo dunque applicare il Master Theorem ed ottenere che la complessità di questo MergeSort modificato è, al caso pessimo, $T(n) = \Theta(n^2)$, dunque la chiamata a QuickSort peggiora l'algoritmo. □

Esercizio 13 (Esame 27/06/2016). 🏷 `Progettazione di Algoritmi` 🏷 `Heap`
Sviluppare e descrivere in pseudocodice un algoritmo che dato uno heap di minimo H e una chiave k, stampi tutte le chiavi in H di valore $\leq k$. La complessità dell'algoritmo deve essere proporzionale al numero m di chiavi stampate, oppure $\Theta(1)$, se $m = 0$.

Soluzione.

```
LessThanK(H, k, node) { // Prima chiamata su (H, k, 1)
   if(node ≤ H.heapsize) {
      if(H[node] ≤ k)
         print(H[node]);
         LessThanK(H, k, left(node));
         LessThanK(H, k, right(node));
   }
}
```

L'algoritmo visita lo heap seguendo la sua struttura ad albero, e terminerà non appena viene trovata una chiave maggiore di k. Questo ci assicura che la complessità dell'algoritmo sarà $T(m) = \Theta(m)$ oppure $\Theta(1)$ se la radice dello heap, in $H[1]$, è subito maggiore di k. La correttezza di questa strategia è assicurata dal fatto che H sia uno heap *di minimo*. □

Esercizio 14 (Esame 12/06/2015). 🏷 `Heap`
Quali sono il numero minimo e il numero massimo di elementi in uno heap di altezza h?

Soluzione.
Poiché uno heap è un albero binario completo per $h-1$ livelli, si ha che il numero di nodi, escluso l'ultimo livello, è pari a $2^{(h-1)+1} - 1$. Questa quantità corrisponde al minimo, in quanto stiamo ignorando l'ultimo livello (altrimenti lo heap non avrebbe avuto altezza h). Per calcolare il massimo, supponiamo di avere tutte le possibili foglie, e quindi di avere un albero binario completo su tutti i livelli, alto h: il numero di nodi in questo caso è pari a $2^{h+1} - 1$. Possiamo quindi concludere che $2^h - 1 \leq \#nodi \leq 2^{h+1} - 1$. □

Esercizio 15 (Esame 12/06/2015 - II). 🏷 `Relazioni di Ricorrenza`
Per un certo problema sono stati trovati due possibili algoritmi risolutivi. Il tempo di esecuzione del primo soddisfa la relazione di ricorrenza:

$$T(n) = 5T(n/6) + 2n^2$$

mentre il secondo algoritmo soddisfa la relazione:

$$T'(n) = 7T'(n/6) + 3n.$$

Si dica, giustificando la risposta, quale algoritmo è da preferire per input di dimensione sufficientemente grande.

Soluzione.
Volendo applicare il Master Theorem, abbiamo che per il primo algoritmo: $a = 5, b = 6, f(n) = 2n^2, n^{\log_b a} = n^{\log_6 5} \approx n^{0.898} = \Omega(n^{(\log_6 5)+\epsilon})$, per $0 \leq \epsilon \leq 2 - \log_6 5$. Ci troviamo dunque nel terzo caso del teorema, e dobbiamo verificare la condizione di regolarità, ovvero $af(n/b) \leq cf(n)$ per $c < 1$. Abbiamo che $5 * 2(\frac{n^2}{36}) \leq 2cn^2$ è soddisfatta per $5/36 \leq c < 1$. Possiamo quindi applicare il teorema ed ottenere che $T(n) = \Theta(n^2)$.
Per il secondo algoritmo si ha: $a = 7, b = 6, f(n) = 3n, n^{\log_b a} = n^{\log_6 7} \approx n^{1.086} = \mathcal{O}(n^{(\log_6 7)-\epsilon})$, per $0 \leq \epsilon \leq (\log_6 7) - 1$. Ci troviamo dunque nel primo caso del Master Theorem e, applicandolo, otteniamo che $T'(n) = \Theta(n^{\log_6 7})$. Poiché $\log_6 7 < 2$, il secondo algoritmo è da preferire per input di dimensione sufficientemente grande. □

Esercizio 16 (Esame 10/09/2015). 🏷 `Heap`
Si consideri uno heap H di massimo e si definisca un algoritmo HEAP-DELETE(H, i) che cancelli la chiave di posizione i ripristinando le proprietà della struttura.

Soluzione.
La strategia da utilizzare è quella di scambiare l'elemento di posizione i con quello in $H.heapsize$, diminuire la dimensione di H e poi ripristinare la proprietà di heap adesso violata dal nuovo nodo i.

```
Heap-Delete(H, i) {
   Scambia(H[i], H[H.heapsize]);
   H.heapsize--;
   while(i > 1 && H[i] > H[parent(i)]) {
      Scambia(H[i], H[parent(i)]);
      i = parent(i);
   }
   /* Se abbiamo eseguito il while, Max Heapify terminerà subito senza fare niente */
   Max-Heapify(H, i);
}
```

La complessità di questo algoritmo è $T(n) = \mathcal{O}(h) = \mathcal{O}(\log n)$, perché lo scambio avviene in tempo costante, e sia il `while` che `Max-Heapify` richiedono tempo pari all'altezza dello heap, che è logaritmica nel numero dei nodi. Si noti che non è corretto chiamare solo heapify in quanto, a seconda del valore che troviamo in $H[H.heapsize]$ potremmo violare la proprietà di heap verso il padre del nodo e non verso i figli, cosa che invece è richiesta per poter applicare heapify. Nel primo caso dunque, bisogna scambiare il nodo con il proprio padre fintanto che il valore contenuto nel nodo è maggiore di quello contenuto nel padre. Infine osserviamo che le due operazioni sono mutuamente esclusive, cioè solo una di esse verrà effettivamente eseguita. □

Esercizio 17 (Esame 11/01/2016). 🏷 `Heap`
Quanto costa la ricerca di una chiave k in uno heap di n elementi al caso pessimo?

Soluzione.
Poiché uno heap (di massimo o di minimo) non gode di particolari proprietà legate alla ricerca, questa si riduce alla scansione dell'array che rappresenta l'albero, per cui $T(n) = \mathcal{O}(n)$. Questo succede perché l'unica cosa che possiamo inferire su ogni nodo dell'albero è che contiene il massimo (o il minimo) del suo sottoalbero, ma non sappiamo come sono disposti gli elementi al suo interno, né ci possiamo accorgere subito se un elemento è presente o no in un dato sottoalbero. □

Esercizio 18 (Esame 11/01/2016 - II). 🏷 `Heap`
Progettare un algoritmo che, dato un array, verifichi *efficientemente* se soddisfa la proprietà di heap. Analizzarne la complessità.

Soluzione.

```
isHeap(A) {
   for i = 1 to ⌊n/2⌋ {
      if(A[i] < A[left(i)])
         return false;
      if(right(i) ≤ n && A[i] < A[right(i)])
         return false;
   }
```

```
   return true;
}
```

$T(n) = \mathcal{O}(n)$. In particolare vengono eseguite soltanto $n/2$ iterazioni del ciclo `for`, il cui corpo richiede tempo costante (si noti che `left(i) = 2i` e `right(i) = 2i + 1`). L'algoritmo procede scendendo nello heap alla ricerca di un nodo i che violi la proprietà di heap di massimo. Non è necessario controllare che `left(i)` $\leq$ `n` poiché `i` varrà al massimo $\lfloor n/2 \rfloor$. □

Esercizio 19 (Esame 03/07/2017). 🏷 `Relazioni di Ricorrenza`
Per un certo problema sono stati trovati due algoritmi risolutivi. Il primo soddisfa la ricorrenza:

$$T_1(n) = 9T_1(n/3) + 2n^2$$

mentre il secondo soddisfa la relazione di ricorrenza:

$$T_2(n) = 3T_2(n/2) + n^2 \log^2 n.$$

Entrambi richiedono tempo $\Theta(1)$ per il caso base. Si dica, giustificando la risposta, quale dei due algoritmi è da preferire per input di grandi dimensioni.

Soluzione.
Volendo applicare il Master Theorem, abbiamo che per il primo algoritmo: $a = 9, b = 3, f(n) = 2n^2$. Poiché $n^{\log_b a} = n^{\log_3 9} = n^2$, ci troviamo nel secondo caso del teorema (infatti $2n^2 = \Theta(n^{\log_3 9})$. Otteniamo quindi che $T_1(n) = \Theta(n^2 \log n)$.
Per il secondo algoritmo si ha: $a = 3, b = 2, f(n) = n^2 \log^2 n$. Poiché $n^{\log_b a} = n^{\log_2 3} \approx n^{1.58}$, ci troviamo nel terzo caso in quanto $n^2 \log^2 n = \Omega(n^{\log_2 3})$. Verifichiamo quindi la condizione di regolarità, cioè troviamo un valore di $c < 1$ tale che $af(n/b) \leq cf(n)$. Sostituendo otteniamo $3\frac{n^2}{4} \log^2 \frac{n}{2} \leq cn^2 \log^2 n$. Osserviamo che per $c = \frac{3}{4} < 1$ la condizione è verificata, infatti vale che $\frac{3}{4} n^2 \log^2 \frac{n}{2} \leq \frac{3}{4} n^2 \log^2 n$. Concludiamo applicando il terzo caso del teorema e ottenendo quindi $T_2(n) = \Theta(n^2 \log^2 n)$.
Per cui è da preferire il primo algoritmo. □

Esercizio 20 (Esame 03/07/2017 - II). 🏷 `Progettazione di Algoritmi`
Dato un array $A[1 \dots n]$ di n interi non necessariamente distinti, si progetti un algoritmo efficiente al caso pessimo che verifica se esistono due indici i e j tali che $A[j] = 2A[i]$ e se ne valuti la complessità.

Soluzione.
```
Doppio(A) {
   B = nuovo array di coppie <elemento di A, posizione in A>
   HeapSort(B); // Rispetto al primo elemento delle coppie
   for i = 1 to n
      j = RicercaBinaria(B.elementi, i, n, 2*B[i].elemento);
      if(j != -1)
         return <B[i].posizione, B[j].posizione>;
   return <-1, -1>; // Non esistono tali i e j
}
```

L'algoritmo crea un array ausiliario di coppie che memorizzano gli elementi di A con la loro vecchia posizione in A, cioè quelle che verranno restituite. Dunque procede ordinando questo nuovo array rispetto alla prima componente delle coppie. Successivamente si procede ricercando per ogni i il valore $2B[i]$ nell'array $B[i \dots n]$ utilizzando la ricerca binaria.

La complessità in tempo di questo algoritmo è quindi $T(n) = T(heapsort)+n*T(binarysearch) = \Theta(n \log n)$, quella in spazio è $S(n) = \Theta(n)$ in quanto utilizziamo l'array B di supporto, lungo n.

Si noti che questa non è una soluzione ottima: è infatti possibile ottenere tempo $\Theta(n)$ utilizzando tecniche più avanzate come l'hashing. □

Esercizio 21. 🏷 `Progettazione di Algoritmi`
Dato un array a di n interi e un intero k, verificare se esiste una coppia di indici $1 \leq i < j \leq n$ tali che $a[i] + a[j] = k$. L'algoritmo non deve richiedere più di $\mathcal{O}(n \log n)$ tempo al caso pessimo.

Soluzione.

```
CercaK(A, k) {
   B = nuovo array di coppie <elemento di A, posizione in A>
   HeapSort(B); // Rispetto al primo elmento delle coppie
   for i = 1 to n
      /* Ricerca Binaria sul primo elemento delle coppie */
      j = RicercaBinaria(B.elementi, i, n, k - B[i].elemento);
      if(j != -1 && B[j].posizione > B[i].posizione)
         return <B[j].posizione, B[i].posizione>;
   return <-1, -1>; // Non esistono tali i e j
}
```

La strategia adottata è quella di ordinare un array di supporto contenente coppie che memorizzano gli elementi di A con la loro vecchia posizione in A. Dunque si procede ordinando questo nuovo array rispetto alla prima componente delle coppie e poi si eseguono ricerche binarie nel sottoarray $B[i \dots n]$ per il valore $k - B[i].elemento = B[j].elemento$, per ogni i finché non viene trovata una soluzione, se esiste.
La complessità di questo algoritmo è $T(n) = T(heapsort) + n * T(ricercabinaria)$ poiché vengono effettuate al più n ricerche binarie. Concludiamo quindi che $T(n) = \Theta(n \log n) + \mathcal{O}(n \log n) = \Theta(n \log n)$, mentre $S(n) = \Theta(n)$ per l'array B di supporto.

Come sopra, anche questa soluzione non è ottima. □