Introduzione agli Algoritmi
Appello esame del 2 luglio 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)
Parte 1

Le risposte non motivate non saranno prese in considerazione.
Negli esercizi di progettazione, prima di passare allo pseudocodice descrivete l'idea algoritmica soggiacente.
Per tutti gli algoritmi progettati è necessario analizzare tempo di esecuzione e correttezza

Esercizio 1

Per ogni affermazione, si scriva se è vera o falsa. La verità di un'affermazione va provata, basandosi sulle definizioni di O, Θ e Ω. La falsità va dimostrata con un contro esempio, cioè fornendo le funzioni che falsificano l'affermazione.

1. $f(n) = \Theta(n)$ e $g(n) = \Omega(n) \Rightarrow f(n)g(n) = \Omega(n^2)$
2. $f(n) = \Omega(n)$ e $g(n) = O(n^2) \Rightarrow f(n)/g(n) = O(n)$
3. $f(n) = O(\log n) \Rightarrow 2^{f(n)} = O(n)$

Sol.
1. L'affermazione è vera. Dalle ipotesi abbiamo che esistono c, c', n_0 e n'_0 tali che $f(n) \geq cn$ e per ogni $n \geq n_0$ e $g(n) \geq c'n$, per ogni $n \geq n'_0$, dunque $f(n)g(n) \geq cc'n^2$, per ogni $n \geq \max\{n_0, n'_0\}$.
2. L'affermazione è falsa. Basta prendere $f(n) = n^4$ e $g(n) = n^2$, in tal caso $f(n)/g(n) = n^2 \neq O(n)$. [3]
3. L'affermazione è falsa. Basta prendere $f(n) = 3\lg n$, allora $2^{3 \lg n} = 2^{\lg n^3} = n^3 \neq O(n)$.

Esercizio 2

Si consideri la seguente funzione:

```
fun (array A, int i, int f) {
        n = f-i+1
        t=n;
        while t≥1 do t =t-2;
        if (n ≤ 1) then return 1;
        else return i + 2*fun (A, i, (i+f)/2);
}
```

Qual è il tempo di esecuzione di *fun* in funzione di n? Si imposti e si risolva la relazione di ricorrenza.

Sol.
Le istruzioni diverse dalla chiamata hanno un tempo di esecuzione in O(n/2), quindi la relazione di ricorrenza è
$T(n) = T(n/2) + cn/2$.
Qui l'albero della ricorsione è degenere
cn/2
cn/4
cn/8
…

Quindi la sommatoria da considerare è $cn(1/2 + (1/2)^2 + (1/2)^3 + \ldots + (1/2)^k)$, dove k è circa il logaritmo di n. Poiché si tratta di un somma di potenze di base minore di 2, possiamo maggiorarla con la sua estensione all'infinito che converge a una costante. Questo ci permette di prevedere che T sia in O(n), cosa che verifichiamo usando l'induzione.
Si tratta di far vedere che esistono b e $n_0 \geq 0$ tali che $f(n) \leq bn$ per ogni $n \geq n_0$. Supponiamo che sia vero per ogni $m < n$ e consideriamo T(n).
Sappiamo che $T(n) = T(n/2) + cn/2 \leq bn/2 + cn/2$, per ipotesi induttiva.
Per determinare se esiste b che soddisfa $T(n) \leq bn$ imponiamolo e vediamo se otteniamo un valore per b o una contraddizione:
$bn/2 + cn/2 \leq bn \iff$ (per $n \geq 1$) $b + c \leq 2b \iff c \leq b$, allora basta prendere $b = c$ e l'equivalenza è vera per ogni $n \geq 1$.
Per la base dell'induzione osserviamo che $T(2) = T(1) + c$, quindi se si prende $b > T(1) + c$, allora si può dire che $T(n) \leq bn$ per ogni $n \geq 1$.
A questo punto si può concludere che il risultato $T(n) = O(n)$ è corretto.

Esercizio 3

Dato un array A di interi si scriva un algoritmo che li riposiziona nell'array, operando **in loco,** in modo tale che i pari siano elementi di indice pari e i dispari siano di indice dispari. Se ci sono più pari che dispari o viceversa questi vanno accumulati alla fine dell'array A.
Per esempio se l'array, con indici che partono da 1, inizialmente è:
A = [50, 47, 92, 78, 76, 7, 60, 36, 59, 30, 50, 43]
il risultato potrebbe essere questo
A = [47, 50, 7, 78, 59, 92, 43, 36, 78, 30, 50, 60]
Si scriva un algoritmo AltPariDisp(A) che in O(n) passi realizza il riposizionamento voluto.

Sol.
Sol. 1 Considerando array che partono da 1, conviene spostare tutti i dispari prima dei pari, con una procedura che ricalca la Partition usata nel quickSort, e poi scambiare i dispari di posto pari con i pari di posto dispari. Se si parte da 0 conviene fare il contrario.
Poiché sia la prima che la seconda fase si concludono in O(n) passi, dove n è il numero degli elementi dell'array, il vincolo di complessità è soddisfatto.
In maggior dettaglio, la prima fase consiste nello scorrimento dell'array con un indice che parte da sinistra (inizialmente da 1) e si incrementa ad ogni passo finché incontra un pari e di un indice che parte da destra (inizialmente da n) e si decremento ad ogni passo finché incontra un dispari, dopo lo scambio del pari a sinistra con il dispari a destra si può far ripartire lo scorrimento da sinistra e da destra su descritto. Questo vuol dire che in un generico momento si avranno gli elementi di indice tra 1 e i-1 tutti dispari e quelli di indice j+1 fino a n tutti pari. Con lo scambio di A[i] con A[j], se $i < j$ e siamo all'interno dell'array, e l'incremento degli indici si ampliano le zone dei pari e dei dispari e il processo ricomincia. Questa procedura dovrà restituire l'indice, ***i***, del primo pari.
Per la seconda fase, si devono considerare alcuni casi che dipendono dal confronto tra il numero dei pari e quello dei dispari.
Caso 1. Se il numero dei dispari, ***i***-1, è maggiore o uguale al doppio di quello dei pari, 2(n-***i***+1), allora tutti i pari andranno scambiati con i dispari di posizione pari a sinistra o uguale a ***i***, a seconda se ***i*** è pari o dispari. Questo perché le posizioni pari a sinistra di i

sono $\lfloor(i-1)/2\rfloor$ e se $\lfloor(i-1)/2\rfloor \geq n-i+1$, bisogna scambiare tutti i pari a destra di i per rispettare il requisito che eventuali elementi dispari in eccesso siano alla fine dell'array.

Esempio:

D	D	D	D	D	D	D	D	D	D	P	P	P	P
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Nella figura D sta per un valore dispari e P per uno pari, qui $i = 11$, e $i-1 = 10 > 2(n-i+1) = 2(14 -11 +1) = 8$. Sistemando tutti pari nei posti pari all'inizio si ottiene:

D	P	D	P	D	P	D	P	D	D	D	D	D	D
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Caso 2. Se il numero dei dispari, $i-1$, è minore o uguale a quello dei pari, $n-i+1$, possiamo scambiare i dispari nelle posizioni pari alla sinistra o uguali a i con i pari nelle posizioni dispari uguali o alla destra di i. Infatti se $i-1 < n-i+1$ c'è modo di sistemare gli $(i-1)/2$ dispari in posizioni pari, nelle posizioni dispari a partire da i o alla sua destra. Gli eventuali pari in eccesso sono correttamente alla fine dell'array.

Esempio:

D	D	D	D	P	P	P	P	P	P	P	P	P	P
1	2	3	4	5	6	7	8	9	10	11	12	13	14

qui $i = 5$, e $i-1 = 4 < n-i+1 = 10$, dopo gli scambi si ha:

D	P	D	P	D	P	D	P	P	P	P	P	P	P
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Caso 3. Se invece $n-i+1 < i-1 < 2(n-i+1)$ c' è un eccesso di dispari alla sinistra di i, perché non ci sono abbastanza pari da intervallare con i dispari, dovendo mantenere l'alternanza anche a destra di i, quindi portiamo i dispari in eccesso in coda. I dispari in eccesso sono $2i - n - 2$. Dopo lo spostamento l'indice del primo pari è diminuito ed è $n - i + 2$. Ora abbiamo lo stesso numero di pari e di dispari alla sinistra dei dispari spostati e possiamo procedere scambiando i dispari nelle posizioni 2,4,…,j, dove j è il più grande pari minore o uguale di $n-i+2$, con i pari nelle posizioni dispari a partire dalla prima uguale o successiva a $n-i+2$ fino alla fine.
Esempio:

D	D	D	D	D	D	D	D	D	D	P	P	P	P	P	P
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Introduzione agli Algoritmi
Prova intermedia 14 Aprile 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

qui ***i*** = 11, e ***i*** -1 = 10 > n-***i*** +1 = 16 - 11 +1 = 6. Gli elementi da scambiare sono ***i***-1 - (n-***i*** +1) = 2 ***i*** - n - 2 = 22-16 -2 = 4. Si scambiano gli ultimi 4 pari con i 4 dispari nelle posizioni ***i*** -1=10,.., ***i*** - 4 +1 =7, ottenendo

D	D	D	D	D	D	P	P	P	P	P	P	D	D	D	D
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Ora, con ***i*** = ***i*** - (2 ***i***-n-2) = n-***i*** +2 = 16 - 11 + 2 = 7, si effettuano gli scambi per ottenere:

D	P	D	P	D	P	D	P	D	P	D	P	D	D	D	D
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Pseudocodice, nell'ipotesi che l'array cominci con indice 1:

```
PartitionPD(A)
Input: un array di interi A
Output: l'array A in cui i dispari precedono i pari, e l'indice del primo pari in A se c'è, altrimenti, cioè se sono tutti dispari, i = n+1.
n = A. length
i=1, j= n
while  (i ≤ j)
//in questo ciclo  se i<j, gli elementi di indice tra 1 e i-1 sono dispari, quelli di indice tra n e j+1 sono pari.
    while  ( i ≤ n and A[i] è dispari) i++  //cerco il primo pari se c'è
    while  ( j ≥ 1 and A[j] è pari) j- -  //cerco il primo dispari se c'è
    if (i<j and i≥n and j≥1)
 //se non sono stati fatti tutti gli scambi, allora  i<j, gli elementi di indice tra i e j sono ancora da considerare, dopo lo scambio e gli incrementi degli indici gli elementi di indice tra 1 e i-1 sono dispari, quelli di indice tra n e j+1 sono pari
          scambia A[i] con A[j]
          i++
          j - -
  return i
Poiché gli elementi da 1 a i-1 sono dispari, i individua il primo pari, se c'è.

ScambiaPariDispari(A,i)
Input un array di interi A e un indice che individua il primo pari, se presente in A, n +1 altrimenti
Output: l'array A con gli elementi dispari di indice dispari e quelli pari di indice pari, eventuali pari o dispari in eccesso in fondo all'array
n = A. length
if i  == n+1 then return // sono tutti dispari
```

Introduzione agli Algoritmi
Appello esame del 2 luglio 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

```
j = 2 // j è l'indice dell'elemento dispari in posizione pari da scambiare, gli scambi si faranno sugli elementi di indice pari tra 2 e i
if i è pari  then k = i+1  else k = i
// k è l'indice dell'elemento pari da scambiare con un dispari di indice pari
if (i-1 ≥ 2(n-i+1) )
    while ( j ≤ n and  k ≤ n and j < i)
            scambia A[k] con A[j]
            k=k+1
            j = j+2
else
    if (n-i+1 < i-1 < 2(n-i+1)  then
        for m = n-i+2 to i-1 to // portiamo i dispari in eccesso in coda
            r = 0
            scambia A[m] con A[n-r]
            r++
        i = n-i+2
        if i è pari  then k = i+1  else k = i // sistemiamo gli indici
// ora si possono scambiare i pari di posto dispari con i dispari di posto pari
  while ( j ≤ n and  k ≤ n and j ≤ i)
          scambia A[k] con A[j]
          k = k+2
          j = j+2
```

Sol. 2 Volendo evitare una doppia scansione dell'array procediamo direttamente.
Per capire come procedere è utile considerare una situazione in cui gli elementi da 1 a i-1 sono già sistemati e chiederci cosa fare con il successivo:

D	P	…	X						
1	2		i						

Supponiamo che anche in i-1 c'è un valore che ha la stessa parità di i-1.

Caso 1.

i è pari

a. se X è pari non c'è alcunché da fare
b. se X è dispari devo cercare un pari alla sua destra, usando un indice p e se e quando lo trovo effettuare lo scambio.

D	P	…	X		Y				
1	2		i		p				

Però se ogni volta dovessi partire dall'indice successivo a quello dell'elemento nel posto sbagliato avrei una complessità quadratica.
Mi chiedo se per una seconda ricerca posso proseguire partendo da p, perché in tal caso non avrei una complessità quadratica, ma solo una doppia scansione dell'array con due indici, e tutto termina quando il primo dei due raggiunge la fine dell'array.
Gli elementi nelle posizioni da i+1 a p - 1 sono tutti dispari, quindi nella posizione i+1 c'è sicuramente un dispari (p è almeno i+1) quindi il primo elemento di cui ci si deve preoccupare è i+2 (se p≥i+2) ma allora la posizione è ancora pari, occupata da un dispari quindi si può far partire la ricerca del pari dall'indice p. Se p fosse stato i+1, si ricomincia con i = i+2, di cui si controlla la parità come prima.

L'altro caso, quando i è dispari è del tutto analogo, e può essere quindi trattato nello stesso modo scambiando pari con dispari e magari utilizzando un altro indice d per la ricerca dei dispari. Il primo indice che raggiunge l'ultimo elemento ferma la ricerca.

Il ragionamento nei due casi è talmente simile che è lecito chiedersi se non si possano unificare.
Questo si può fare utilizzando una funzione pari(x) che dà 1 se x è pari e 0 se è dispari.

Allora il test diventa: pari(A[i]) = pari(i). Se la risposta è sì non si fa niente, altrimenti, cioè se pari(A[i]) ≠ pari(i), si deve cercare un elemento A[j] tale che pari(A[j]) ≠ pari(A[i]), perché in tal caso sarà pari(A[j]) = pari(i).
Si scorre l'array alla ricerca di j a partire da i +1 e incrementandolo ogni volta che pari(A[j]) = pari(A[i]). Se e quando j viene trovato si può eseguire lo scambio. Si può di nuovo ripartire da j per le ricerche successive di elementi da scambiare?
La risposta è sì, perché, come prima, gli elementi di indice i+1 fino a j-1 hanno la stessa parità di A[i] prima dello scambio, quindi A[i+1] è sicuramente a posto, il problema si pone con A[i+2], se j≥i+2, ma per A[i+2] si deve fare ancora una ricerca di un elemento della parità di A[i] prima dello scambio, dunque si può far partire la ricerca da j+1.
Se invece j=i+1, i viene incrementato di 2 e comincia una fase analoga con ricerca dell'elemento, se necessario, a partire da j= i+3.
Se successivamente si ripropone il problema di cercare un elemento da scambiare di parità diversa risulterà j < i e j dovrà quindi essere inizializzato a i+1.
Poiché il controllo sulla disposizione corretta degli elementi procede fino a quando si trova un elemento da sostituire a quello in posizione scorretta, si può concludere che in coda all'array si troveranno tutti gli eventuali dispari o pari in eccesso.

Pseudocodice:

```
AltPariDispari(A)
n = A.length
i=1, j=0
while  (i≤ n) do
  if (pari(A[i]) ==  pari(i)) then i++
  else
     if (i > j) then  j = i+1 else  j = j+1
```

```
while ( (j≤n)  and ( pari(A[j]) == pari(A[i]) ) j++
if j≤ n then
      scambia  A[j] e A[i]
      i = i+2
else i = n+1
```

Esempio:

D	D	D	D	D	D	D	D	D	D	P	P	P	P	P	P
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Qui gli scambi sono tra A[2] e A[11], A[4] e A[12], A[6] e A [13], A[8] e A[14], A[10] e A[15] e producono il seguente risultato:

D	P	D	P	D	P	D	P	D	P	D	D	D	D	D	P
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

poi viene considerata l'entrata 12 che viola la proprietà e quindi si effettua lo scambio, con A[16], producendo:

D	P	D	P	D	P	D	P	D	P	D	P	D	D	D	D
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

A questo punto i vale 14, ma j assume il valore 17 e si esce dal ciclo.

Parte II

Le risposte non motivate non saranno prese in considerazione.
Negli esercizi di progettazione, prima di passare allo pseudocodice descrivete l'idea algoritmica soggiacente.
Per tutti gli algoritmi progettati è necessario analizzare tempo di esecuzione e correttezza.

Esercizio 4

Si dimostri che un AVL con n nodi ha altezza O(lg n). La dimostrazione deve essere completa in tutti i suoi passaggi ed autocontenuta.

Sol. Si veda:
lucidi lezione o
http://interactivepython.org/runestone/static/pythonds/Trees/AVLTreePerformance.html

Esercizio 5

Un insieme di chiavi è memorizzata in un Max-heap e in un ABR T.
Quale delle due strutture dati è più efficiente se vogliamo stampare le chiavi in ordine decrescente?
Si scrivano i due algoritmi e li si analizzino dal punto di vista del tempo di esecuzione asintotico.

Introduzione agli Algoritmi
Prova intermedia 14 Aprile 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

Sol. Per stampare in ordine decrescente le chiavi memorizzate in un ABR basta effettuare una visita inorder rovesciata, cioè visitando prima il sotto albero destro, poi la radice e poi il sotto albero sinistro.
Ogni visita ha un costo lineare nel numero dei nodi, quindi se n è il numero delle chiavi si ha un tempo di esecuzione in $\Theta(n)$.
Per stampare in ordine decrescente le chiavi memorizzate in un max heap, basta richiamare n volte la funzione extract-max stampando di volta in volta il risultato ottenuto. Poiché extract-max ha un tempo di esecuzione in O(lg n) questo modo di procedere comporta O(n lg n) passi. E' quindi più conveniente per questo problema partire da un ABR.
Per le implementazioni si veda il libro di testo.

Esercizio 6

Si scriva un algoritmo TerTreeSearch(T,k) che prende in input la radice di un albero ternario T e una chiave k, restituendo il nodo che ha la chiave k o NIL se la chiave non è presente. Non ci sono chiavi duplicate in T.
Un albero ternario è concettualmente identico ad un albero binario ma ogni nodo x ha al più tre figli x.left, x.right e x.center e ha due chiavi x.key1 e x.key2, con x.key1 < x.key2. I nodi nel sotto albero sinistro di x contengono chiavi k1 e k2 tali che k1 < k2 < x.key1, i nodi nel sotto albero destro k1 e k2 tali che x.key2 < k1 < k2 e quelle nel sotto albero centrale le chiavi k1 e k2 tali che x.key1 < k1 < k2 < x.key2.
Qual'è la complessità di tempo asintotica dell'operazione di ricerca?

Per esempio, la ricerca della chiave 61 deve restituire il puntatore al nodo che ha le chiavi 61,65

Sol. Per risolvere questo problema dobbiamo, come nel caso degli alberi binari di ricerca , sfruttare la proprietà di ordinamento sui dati. In questo caso così semplice lo pseudocodice è sufficiente a descrivere l'algoritmo:

```
TerTreeSearch(T,k)
if (T == NULL  or T.key1 == k or  T. key2  == k)  returnT
if ( k < T.key1) TerTreeSearch(T.left,k)
if ( k > T.key2) TerTreeSearch(T.right,k)
else TerTreeSearch(T.center,k)
```

Introduzione agli Algoritmi
Appello esame del 2 luglio 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

La complessità è O(h) visto che nel caso peggiore si scende di padre in figlio ad ogni passo fino a raggiungere una foglia di profondità massima.