Esercizi ragionati relativi all'insegnamento di Introduzione agli Algoritmi (CdL in Informatica)

Prof.ssaTiziana Calamoneri e Prof. Angelo Monti

Questi esercizi ragionati si riferiscono all'insegnamento di Introduzione agli Algoritmi per il Corso di Laurea in Informatica, e vanno associati alle relative dispense, di cui vengono richiamati i numeri dei capitoli.

Questi esercizi rispecchiano piuttosto fedelmente il livello di dettaglio che viene richiesto durante l'esame scritto, ma allo stesso tempo, vogliono condurre lo studente verso il corretto modo di ragionare, per cui possono sembrare a volte prolissi.

Sono benvenuti esercizi svolti dagli studenti, che verranno uniformati a questi ed annessi al presente documento.

Questo testo vuole essere ad **alta leggibilità**. Nel passaggio a questa modalità potremmo avere introdotto degli errori; se doveste trovarne, per favore segnalateli. Inoltre, ogni suggerimento volto a migliorare la leggibilità di questo documento è benvenuto.

Ogni tipo di contributo può essere inviato a:
`calamo@di.uniroma1.it` e `monti@di.uniroma1.it`.

1 Esercizio.

Riferimento ai capitoli: 1. Introduzione; 2. Notazione asintotica

Si consideri il seguente programma:

```
def somma(n):
    s=0
    for i in range(1,n+1):
        s+= i
    return s
```

e si calcoli il costo computazionale asintotico nel caso peggiore.

Si dica, inoltre, se l'algoritmo descritto dal codice dato è efficiente per il problema che esso si prefigge di risolvere.

Soluzione.

Come primo passo, dobbiamo riconoscere quale sia il parametro che guida la crescita del costo computazionale. In casi come questo, in cui compare un elenco di n elementi, è immediato definire n come parametro. Quindi il costo sarà una funzione $T(n)$.

Secondo il modello RAM, una riga i può richiedere un tempo diverso da un'altra, ma ognuna impiega un tempo costante c_i. Poiché il ciclo viene ripetuto n volte (e la sua condizione testata n + 1 volte), si ha:

$$T(n) = c_1 + c_2\ (n + 1) + nc_3 + c_4 = an + b$$

per opportuni valori di a e b. Ne consegue che $T(n) = \Theta(n)$.

È immediato riconoscere che lo pseudocodice descrive l'algoritmo di somma dei primi n interi. Tuttavia, esso non è l'algoritmo più efficiente. Si consideri, infatti, il seguente programma:

```
def somma(n):
    return n*(n+1)//2
```

il cui costo computazionale è, ovviamente $T(n) = c_1 = \Theta(1)$.

È interessante notare che, nella soluzione precedente, non abbiamo parlato di *dimensione dell'input* ma di *parametro che guida la crescita del costo*. In effetti, tenendo conto che l'input è un singolo intero, la quantità di memoria necessaria è costante o, tutt'al piú, proporzionale a $\log_2 n$ (numero di bits necessari per memorizzare il valore n). Affinché un algoritmo sia efficiente, il suo costo computazionale deve essere polinomiale nella dimensione dell'input, cioè $O(\log n)^c)$, per una qualche costante c. Invece, nel nostro caso, il costo è $O(n) = O(2^{\log n})$. Viceversa, la soluzione che lavora in tempo costante può essere considerata efficiente perchè polinomiale nella dimensione dell'input.
Infine, osserviamo che, se assumiamo che la dimensione dell'input è logaritmica e non costante, sarebbe allora più corretto porsi nella misura di costo logaritmico e non uniforme, ed in tal caso il costo è $O(\log^2 n)$.

2 Esercizio.

Riferimento ai capitoli: 5. Equazioni di ricorrenza

Si risolva la seguente equazione di ricorrenza con due metodi diversi, per ciascuno dettagliando il procedimento usato:

$$\begin{cases} T(n) = T\left(\frac{n}{2}\right) + \Theta\left(n^2\right) \text{ se } n > 1 \\ T(1) = \Theta(1) \end{cases}$$

tenendo conto che *n* è una potenza di 2.

Soluzione.

Utilizziamo dapprima il metodo principale, che è il più rapido, e poi verifichiamo la soluzione trovata con un altro metodo.

Le costanti *a* e *b* del teorema principale valgono qui 2 e 1, rispettivamente, quindi $\log_b a = 0$ ed $n^{\log_b a} = 1$, per cui, ponendo $\epsilon < 2$, siamo nel caso 3. del teorema, se vale che $af(n/b) \leq cf(n)$.

Poiché f(n) è espressa tramite notazione asintotica, per verificare la disuguaglianza, dobbiamo eliminare tale notazione, e porre ad esempio:
$f(n) = \Theta(n^2) = hn^2 + k$.

Non è restrittivo supporre $h, k \geq 0$ visto che ci troviamo in presenza di un costo computazionale e quindi non è sensato avere tempi di esecuzione negativi.

Ci chiediamo se esista una costante *c* tale che $h(n/2)^2 + k \leq c(hn^2 + k)$.

Risolvendo otteniamo: $hn^2(1/4 - c) + k(1 - c) \leq 0$ che è verificata ad esempio per $c = 1/2$.

Ne possiamo dedurre che $T(n) = \Theta(n^2)$.

Come secondo metodo usiamo quello iterativo. Dall'equazione di ricorrenza deduciamo che:

$$T(\tfrac{n}{2}) = T(\tfrac{n}{2^2}) + \Theta((\tfrac{n}{2})^2) \text{ e } T(\tfrac{n}{2^2}) = T(\tfrac{n}{2^3}) + \Theta((\tfrac{n}{2^2})^2)$$

e così via. Sostituendo:

$$
\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + \Theta(n^2) \\
&= T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) \\
&= T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) \\
&\ldots \\
&= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right)
\end{aligned}
$$

Continuiamo ad iterare fino al raggiungimento del caso base, cioè fino a quando $n/2^k = 1$, il che avviene se e solo se $k = \log n$.

L'equazione diventa così:

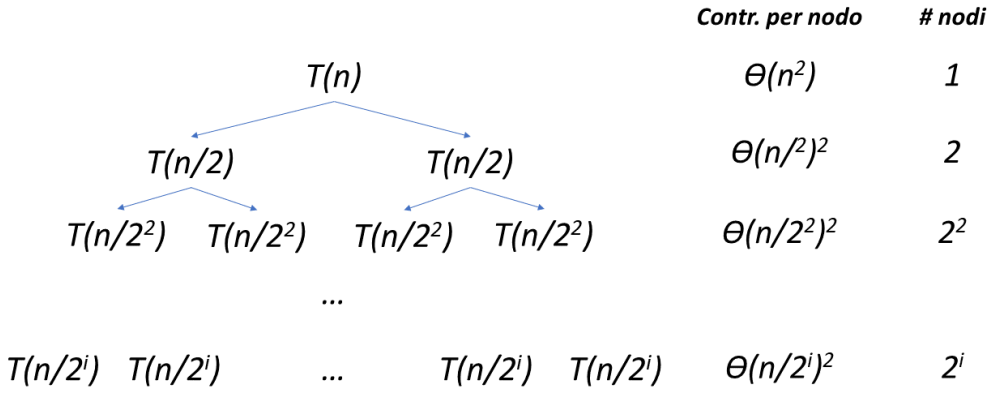
$$T(n) = T(1) + \sum_{i=0}^{\log n - 1} \Theta((\frac{n}{2^i})^2) = \Theta(1) + n^2 \sum_{i=0}^{\log n - 1} \Theta(\frac{1}{4^i}).$$

Ricordando che $\sum_{i=0}^{b} a^i = \frac{a^{b+1}-1}{a-1}$, otteniamo infine:

$$T(n) = \Theta(1) + n^2\Theta\left(\frac{1 - \left(\frac{1}{4}^{\log n}\right)}{1 - \frac{1}{4}}\right) = \Theta(n^2).$$

Anche se forse superfluo, concludiamo con l'osservare che i due metodi ***devono*** portare allo stesso risultato o, quanto meno a risultati compatibili, altrimenti bisogna concludere che si è fatto un errore.

Anche se non richiesto, risolviamo l'equazione anche con il metodo dell'albero:



Poiché ad ogni passo suddividiamo l'input a metà, l'altezza dell'albero sarà $\log_2 n$ e quindi:

$$T(n) = \sum_{i=0}^{\log n} 2^i \Theta(\frac{n}{2^i})^2 = \Theta(n^2) \sum_{i=0}^{\log n} \frac{1}{2^i} = \Theta(n^2).$$

per ragioni simili alle precedenti che quindi non dettagliamo di nuovo qui.

3 Esercizio.

Riferimento ai capitoli: 5. Equazioni di ricorrenza

Si risolva la seguente equazione di ricorrenza con tutti e quattro i metodi, per ciascuno dettagliando il procedimento usato:

$$\begin{cases} T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + \Theta(n) \text{ se } n > 1 \\ T(1) = \Theta(1) \end{cases}$$

Soluzione.

Osserviamo dapprima che il metodo principale non può essere utilizzato, in quanto l'equazione di ricorrenza non è del tipo $T(n) = aT(n/b) + f(n)$.

Procediamo con il metodo iterativo. Si vede subito che i calcoli diventano subito difficili da gestire, quindi utilizziamo la disuguaglianza $T(n/3) \leq T(2n/3)$, sperando di arrivare ad una limitazione inferiore e ad una superiore che coincidano in termini asintotici.

Maggiorando e minorando rispettivamente, ottenendo le due disequazioni:

$$\begin{array}{lll} T(n) \leq 2T(2n/3) + \Theta(n) & \text{e} & T(n) \geq 2T(n/3) + \Theta(n) \\ T(1) = \Theta(1) & & T(1) = \Theta(1). \end{array}$$

Come verifica preliminare, possiamo applicare il metodo principale a queste due ricorrenze, ottenendo che $T(n) = O(n^{\log_{3/2} 2})$ e $T(n) = \Omega(n)$; in effetti $\log_{3/2} 2$ è circa 1.7 quindi sfortunatamente le due limitazioni non coincidono, anche se sono piuttosto vicine. Ovviamente, il metodo iterativo dovrà darci gli stessi risultati. Vediamo:

$$\begin{aligned} T(n) &\leq 2T\left(\frac{2n}{3}\right) + \Theta(n) \leq 2\left(2T\left(\frac{2^2 n}{3^2}\right) + \Theta\left(\frac{2n}{3}\right)\right) + \Theta(n) = \\ &= 2^2 T\left(\frac{2^2 n}{3^2}\right) + 2\Theta\left(\frac{2n}{3}\right) + \Theta(n) \leq \ldots \\ &\leq 2^k T\left(\frac{2^k n}{3^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{2^i n}{3^i}\right) = 2^k T\left(\frac{2^k n}{3^k}\right) + \Theta(n) \sum_{i=0}^{k-1} \left(\frac{4}{3}\right)^i \leq \ldots \end{aligned}$$

finchè $\frac{2^k n}{3^k} = 1 \Leftrightarrow k = \log_{3/2} n$.
Otteniamo dunque:

$$T(n) \leq 2^{\log_{3/2} n} T(1) + \Theta(n) \sum_{i=0}^{\log_{3/2} n - 1} \left(\frac{4}{3}\right)^i =$$

$$= \left(\frac{3}{2}\right)^{\log_{3/2} 2 \log_{3/2} n} \Theta(1) + \Theta(n)\Theta\left(\left(\frac{4}{3}\right)^{\log_{3/2} n}\right) =$$

$$= \Theta\left(n^{\log_{3/2} 2}\right) + \Theta(n)\Theta\left(2^{\log_{3/2} n}\left(\frac{2}{3}\right)^{\log_{3/2} n}\right) =$$

$$= \Theta\left(n^{\log_{3/2} 2}\right) + \Theta(n)\Theta\left(\left(\frac{3}{2}\right)^{\log_{3/2} 2 \log_{3/2} n}\left(\frac{1}{\left(\frac{3}{2}\right)}\right)^{\log_{3/2} n}\right) =$$

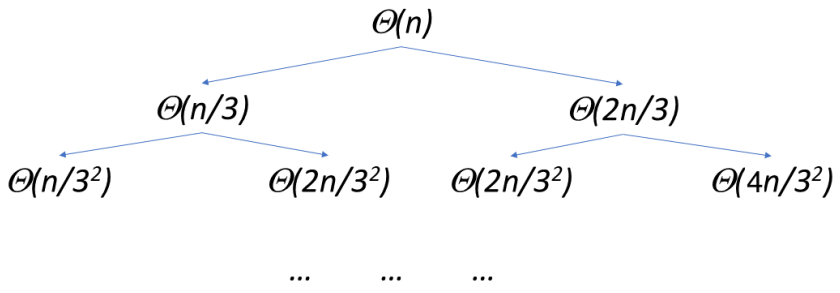
$$= \Theta\left(n^{\log_{3/2} 2}\right) + \Theta(n)\Theta\left(n^{\log_{\frac{3}{2}} 2} \cdot \frac{1}{n}\right) = \Theta\left(n^{\log_{3/2} 2}\right).$$

Da qui deduciamo che $T(n) = O(n^{\log_{3/2} 2})$.
Consideriamo ora l'altra diseguaglianza:

$$T(n) \geq 2T\left(\frac{n}{3}\right) + \Theta(n) \geq ... \geq 2^k T\left(\frac{n}{3^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{n}{3^i}\right)$$

finché $n/3^k = 1 \Leftrightarrow k = \log_3 n$.
Otteniamo dunque:
$T(n) \geq 3^{\log_3 2 \ \log_3 n}\Theta(1) + \Theta(n) \sum_{i=0}^{\log_3 n - 1} \left(\frac{2}{3}\right)^i = \Theta\left(n^{\log_3 2}\right) + \Theta(n) = \Theta(n)$.
Da qui deduciamo che $T(n) = \Omega(n)$.

Utilizziamo ora il metodo dell'albero per vedere se riusciamo ad ottenere un limite asintotico stretto. Si può costruire un albero binario la cui parte superiore è così fatta:



È abbastanza semplice rendersi conto che il contributo di ogni livello è sempre lo stesso, pari a $\Theta(n)$. Cerchiamo di valutare l'altezza di questo albero. Purtroppo non siamo di fronte ad un albero binario completo ed, in particolare, il ramo sinistro sarà più corto di quello destro, infatti il numero di livelli nel ramo sinistro è $log_3 n$ mentre il numero di livelli nel ramo destro è $log_{3/2} n$; ogni ramo intermedio avrà un numero di livelli compreso tra i due.

Per questo, dobbiamo di nuovo maggiorare e minorare il valore di T(n). Ricordando che il contributo di ciascun livello è sempre $\Theta(n)$, abbiamo:

$$\log_3 n\Theta(n) \leq T(n) \leq \log_{3/2} n\Theta(n).$$

Se utilizziamo la formula per la trasformazione della base dei logaritmi $\log_a n = \log_b n \log_a b$ otteniamo che sia il primo che il terzo membro della precedente catena di disuguaglianze sono pari a $\Theta(n \log n)$, per cui in questo caso siamo riusciti a trovare due limitazioni coincidenti, che ci consentono di valutare in senso stretto la soluzione della ricorrenza. Osserviamo che questa soluzione è compatibile con le limitazioni inferiore e superiore ottenuti con il metodo iterativo.

Risolviamo, infine, con il metodo di sostituzione, dapprima eliminando la notazione asintotica dall'equazione di ricorrenza:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \text{ se } n > 1 \text{ e } T(1) = d$$

e poi ipotizzando una soluzione. Visto ciò che abbiamo ottenuto con il metodo dell'albero, possiamo ipotizzare $T(n) \leq an \log n$, dove *a* è una

costante da determinare. Poichè quando n = 1, $\log n = 0$, dobbiamo recuperare un altro caso base.
Consideriamo n = 2; in tal caso T(n) = 2d + 2c = d′, per cui aggiungiamo il caso base T(2) = d′.

Ora procediamo per induzione.

Passo base: Se n = 2 abbiamo $d' = T(2) \leq 2a \log 2 = 2a$; questa relazione impone che la disuguaglianza sia verificata se $a \geq d'/2$.
Passo induttivo: Se $n > 2$, sfruttando l'ipotesi induttiva, possiamo scrivere:

$$
\begin{aligned}
T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn &\leq a\frac{n}{3}\log\frac{n}{3} + a\frac{2n}{3}\log\frac{2n}{3} + cn = \\
&= a\frac{n}{3}\log n - a\frac{n}{3}\log 3 + a\frac{2n}{3}\log n + a\frac{2n}{3}\log\frac{2}{3} + cn = \\
&= an\log n - a\frac{n}{3}\log 3 + a\frac{2n}{3}\log 2 - a\frac{2n}{3}\log 3 + cn = \\
&= an\log n - an\log 3 + a\frac{2n}{3}\log 2 + cn < an\log n - an\log 2 + a\frac{2n}{3}\log 2 + cn = \\
&= an\log n - a\frac{n}{3}\log 2 + cn = an\log n - n\left(\frac{a}{3}\log 2 - c\right) \leq \\
&\leq an\log n \text{ per } a \geq \frac{3c}{\log 2}.
\end{aligned}
$$

Deduciamo che la nostra ipotesi è vera se $a \geq \frac{3c}{\log 2}$. Abbiamo così dimostrato che T(n) = O(nlogn).

Ipotizziamo ora la soluzione $T(n) \geq sn\log n + tn$ per qualche costante *s* e *t* da determinare.

Anche in questo caso, procediamo per induzione.

Passo base: Se n = 1, allora $d = T(1) \geq t$; la disuguaglianza è verificata se $t \leq d$.

Passo induttivo: Se $n > 1$, sfruttando l'ipotesi induttiva ed evitando di ripetere alcuni passaggi già dettagliati nel caso precedente, possiamo

scrivere:

$$
\begin{aligned}
T(n) &= T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \geq \\
&\geq s\frac{n}{3}\log n - s\frac{n}{3}\log 3 + t\frac{n}{3} + 2s\frac{n}{3}\log n - 2s\frac{n}{3}\log\frac{3}{2} + 2t\frac{n}{3} + cn = \\
&= sn\log n - s\frac{n}{3}\left(\log 3 + 2\log\frac{3}{2}\right) + tn + cn \geq sn\log n - s\frac{n}{3}(2+2) + tn + cn = \\
&\qquad\qquad \text{(abbiamo usato che } \log 3 < 2 \text{ e che } \log\frac{3}{2} < 1\text{)} \\
&= sn\log n + n\left(t - \frac{4s}{3}\right) + cn \geq sn\log n \text{ se } t > \frac{4s}{3}.
\end{aligned}
$$

Deduciamo che la nostra ipotesi è vera se *s* e *t* sono tali che $t > \frac{4s}{3}$. Abbiamo così dimostrato che$T(n) = \Omega(n\log n)$.

Dai due risultati parziali concludiamo infine che $T(n) = \Theta(n\log n)$.

4 Esercizio.

Riferimento ai capitoli: 2. Notazione asintotica; 5. Equazioni di ricorrenza

Si consideri il seguente programma:

```
    def Analizzami(A,n):
1       if n<= 3: return A[0]
2       j= 1
3       while j < n:
4           A[j] = A[j] - A[n-j]
5           j *= 3
6       for i in range(3):
7           A[i] += A[i+1]
8           Analizzami(A, n//3)
```

e si mostri tramite il metodo iterativo che il costo computazionale asintotico è $O(n \log n)$.

Soluzione.

Dobbiamo valutare il costo computazionale di ogni singola operazione. Detto T(n) il costo della funzione richiamata su un array con *n* elementi:

- l'istruz. 1. è costante e corrisponde al caso base;
- l'istruz. 2. è costante;
- Cerchiamo di capire quante viene eseguito il ciclo while dell'istruz. 3: *j* parte da 1 e si triplica ad ogni iterazione; quindi, il numero di iterazioni (che si concludono quando *j* raggiunge o supera *n*), sarà $\log_3 n$. Ogni iterazione poi (istruz. 4. e 5.) costa $\Theta(1)$;
- Le istruzioni 7. (costo $\Theta(1)$) e 8. (costo $T(n/3)$) vengono eseguite 3 volte a causa del ciclo for dell'istruzione 6.

Riassumendo: $T(n) = \Theta(1)$ se $n \leq 3$ e $T(n) = 3T(n/3) + \Theta(\log_3 n)$ altrimenti.

Per risolvere questa equazione di ricorrenza dobbiamo usare il metodo iterativo, con cui abbiamo:

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + \Theta(\log_3 n) = \\ &= 3\left(3T\left(\frac{n}{3^2}\right) + \Theta\left(\log_3 \frac{n}{3}\right)\right) + \Theta(\log_3 n) = \\ &= 3^2 T\left(\frac{n}{3^2}\right) + 3\Theta\left(\log_3 \frac{n}{3}\right) + \Theta(\log_3 n) = \ldots = \\ &3^k T\left(\frac{n}{3^k}\right) + \sum_{i=0}^{k-1} 3^i \Theta\left(\log_3 \frac{n}{3^i}\right). \end{aligned}$$

Iterando finchè $\frac{n}{3^k} = 1$ e cioè fino a quando $k = \log_3 n$ si ha:

$$\begin{aligned} T(n) &= 3^{\log_3 n} T(n) + \sum_{i=0}^{\log_3 n - 1} 3^i \Theta\left(\log_3 \frac{n}{3^i}\right) = n\Theta(1) + \sum_{i=0}^{\log_3 n - 1} 3^i \Theta\left(\log_3 n - \log_3 3^i\right) = \\ &= n\Theta(1) + \Theta(\log_3 n) \sum_{i=0}^{\log_3 n - 1} 3^i - \sum_{i=0}^{\log_3 n - 1} \Theta(i) = \\ &= n\Theta(1) + \Theta(\log_3 n) 3^{\log_3 n} - \Theta((\log_3 n)^2) = \Theta(n \log n). \end{aligned}$$

L'ultimo passaggio è stato possibile sfruttando la formula di cambio di base del logaritmo, secondo la quale $\log_2 n$ e $\log_3 n$ differiscono per una costante moltiplicativa.

Essendo $T(n) \leq \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$, si ha che $T(n) = O(n \log n)$.

Ci chiediamo ora quanto sia il risultato in termini di ordine asintotico stretto. Ci viene in aiuto la risoluzione tramite teorema principale. Infatti, $n^{\log_b a} = n^{\log_3 3} = n$, ed $f(n) = O(n^{1-\epsilon})$, prendendo, ad esempio, $\epsilon = 1/2$ e ricordando che il logaritmo cresce più lentamente della radice quadrata. Allora siamo nel primo caso del teorema principale, e perciò la soluzione è

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n).$$

Cerchiamo ora di capire come ottenere questo risultato anche con il metodo iterativo applicato prima. Innanzi tutto, è subito chiaro che non potremo sfruttare la notazione asintotica ma dovremo essere più precisi.

Partiamo da qui:

$$T(n) = 3^{log_3 n} T(1) + \sum_{i=0}^{\log_3 n - 1} 3^i \Theta(\log_3 \frac{n}{3^i}) = \Theta(n) + \Theta(\sum_{i=0}^{\log_3 n - 1} 3^i \log_3 \frac{n}{3^i})$$

e valutiamo la sommatoria:

$$\sum_{i=0}^{\log_3 n-1} 3^i \log_3 \frac{n}{3^i} = \sum_{i=0}^{\log_3 n-1} 3^i \log_3 n - \sum_{i=0}^{\log_3 n-1} 3^i \log_3 3^i =$$
$$= \log_3 n \sum_{i=0}^{\log_3 n-1} 3^i - \log_3 3 \sum_{i=0}^{\log_3 n-1} i\, 3^i$$

avendo sfruttato la seguente proprietà dei logaritmi: $\log_c a^b = b \log_c a$. Ricordiamo il noto risultato $\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1}$ valido per ogni $a \neq 1$ e otteniamo che:

$$\sum_{i=0}^{\log_3 n - 1} 3^i = \frac{3^{\log_3 n} - 1}{2} = \frac{n-1}{2}.$$

Per quanto riguarda l'altra sommatoria, sfruttiamo il seguente risultato: $\sum_{i=0}^{k} i\, a^i = \frac{a^{k+1}(ak-k-1)+a}{(a-1)^2}$ valido per ogni $a \neq 1$ e otteniamo:

$$\sum_{i=0}^{\log_3 n - 1} i\, 3^i = \frac{2\, n \log_3 n - 3n + 3}{4}.$$

Sommando i due contributi abbiamo:

$$\log_3 n \sum_{i=0}^{\log_3 n - 1} 3^i - \log_3 3 \sum_{i=0}^{log_3 n - 1} 3\, 3^i = \log_3 n\, (\frac{n-1}{2}) - \frac{2\, n \log_3 n - 3\, n + 3}{4} =$$

$$= \frac{2\, n \log_3 n - 2 \log_3 n - 2\, n \log_3 n + 3\, n - 3}{4} = \frac{3}{4} n - \frac{\log_3 n}{2} - \frac{3}{4} = \Theta(n).$$

5 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dati in input *n* e *k* interi positivi, si scrivano due funzioni, una iterativa ed una ricorsiva che restituiscano in output l'intero n^k. Si studi poi il costo computazionale delle due funzioni.

Soluzione.

Le due funzioni sono entrambe molto semplici, e possono essere scritte come segue:

```
    def potenza_ite(n,k):
1       if k==0: return 1
2       aux=1
3       for i in range(k):
4           aux=n*aux
5       return aux


    def potenza_ric(n,k):
1       if k==0: return 1
2       return n*potenza_ric(n,k-1)
```

Per calcolare il costo computazionale, individuiamo dapprima i parametri della funzione che, ragionevolmente, saranno *n* e *k*, quindi cerchiamo una funzione *T(n,k)*.

Cominciamo dalla funzione iterativa, che non richiede particolari strumenti: le linee 1., 2. e 5. hanno costo costante; il ciclo della linea 3. è ripetuto esattamente *k* volte e contiene, al suo interno, una sola istruzione che viene eseguita in tempo costante. Pertanto, $T(n, k) = \Theta(1) + \Theta(k) = \Theta(k)$, deducendo così che la funzione *T(n,k)* è in realtà, semplicemente *T(k)*.

Per calcolare il costo computazionale della funzione ricorsiva, osserviamo che la linea 1. richiede tempo costante e costituisce il caso base della

ricorsione. La linea 2., invece, è il cuore della ricorsione ed ha costo $\Theta(1)+$ $T(n, k-1)$. Abbiamo pertanto:

$$\begin{cases} T(n,0) = \Theta(1) \\ T(n,k) = \Theta(1) + T(n,k-1) \text{ se } n > 1. \end{cases}$$

Risolvendo questa equazione di ricorrenza, ad esempio, tramite il metodo iterativo, otteniamo:

$$T(n,k) = \Theta(1)+T(n,k-1) = \Theta(1)+\Theta(1)+T(n,k-2) = \cdots = j\Theta(1)+T(n,k-j).$$

Procediamo a svolgere l'equazione fino a quando non giungiamo al caso base, cioè fino a che $j = k$. In tal caso si ha:

$$T(n,k) = kT(n,k-k) = k\Theta(1) = \Theta(k).$$

Cioè, per entrambe le versioni otteniamo lo stesso costo computazionale.

Si osservi che con un criterio di misura diverso (non più di costo uniforme ma di costo logaritmico), effettivamente il costo computazionale è funzione sia di k che di n, poichè ad esempio ciascuna moltiplicazione per n non impiega tempo costante ma funzione almeno di $\log n$. Per semplicità, non approfondiamo ulteriormente questo argomento.

6 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dati in input un intero n ed un array $A=(a_0, \dots, a_{n-1})$ di n numeri interi, si scrivano due funzioni, una iterativa ed una ricorsiva, che restituiscano in output la somma $a_0 + a_1 + \cdots + a_{n-1}$. Si studi poi il costo computazionale delle due funzioni.

Soluzione.

Questo esercizio è del tutto analogo al precedente, e pertanto possiamo essere più stringati nelle spiegazioni.

Le due funzioni possono essere scritte come segue:

```
    def Somma_ite(A,n):
1       aux=0
2       for i in range(n):
3           aux+= A[i]
4       return aux


    def Somma_ric(A,n):
1       if n==0: return 0
2       return A[n-1]+Somma_ric(A,n-1)
```

Qui il parametro del costo computazionale è ovviamente n; nel caso iterativo esso è facilmente$T(n) = \Theta(1) + n\Theta(1) = \Theta(n)$ mentre nel caso ricorsivo si ha:

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

che è analoga all'equazione di ricorrenza incontrata nel precedente esercizio e risolta dà $T(n) = \Theta(n)$.

7 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dati in input un intero n ed un array $A=(a_0, \ldots, a_{n-1})$ di n numeri reali, si scrivano due funzioni, una iterativa ed una ricorsiva, che restituiscano il valore 1 se il array è palindromo, il valore 0 altrimenti.

Soluzione.

Verificare se un array è palindromo significa domandarsi se a_0 e a_{n-1}, a_1 e a_{n-2}, ecc. hanno a coppie lo stesso valore. l'unica piccola difficoltà che può dunque presentare questo esercizio è la gestione degli indici.

La funzione ricorsiva può essere costruita riportandosi al problema di dimensione inferiore che si ottiene eliminando dall'array il primo e l'ultimo elemento. Sappiamo che, decrementando il valore di n, possiamo ignorare la parte finale dell'array; introduciamo una nuova variabile m che è, inizialmente, settata a 0, ed indica il primo elemento da considerare nell'array. Cioè:

```
    def Palindromo_Ric(A, n, m)
1       if n-1<=m return 1
2       if A[m]==A[n-1])
3           return Palindromo_Ric(A, m+1, n-1)
4       return 0
```

Si osservi che la condizione della linea 1, corrispondente al caso base, è verificata solo se non si è mai passati per la linea 3, cioè se fino ad ora tutti i controlli hanno dato esito positivo. È chiaro, quindi che in tal caso la funzione restituisca 1, indicando cioè che l'array nullo è, ovviamente, palindromo. Nel caso generale, se una coppia viene trovata non rispondente alla definizione di array palindromo, la funzione restituisce 0 senza proseguire con inutili controlli.

Il costo computazionale di questa funzione si può calcolare grazie alla seguente equazione di ricorrenza:

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

che si risolve analogamente ai casi precedenti, ottenendo $T(n) = n/2\Theta(1) + T(0) = \Theta(n/2) = \Theta(n)$. Tenendo conto del fatto che in effetti la funzione ricorsiva non viene richiamata esattamente n/2 volte ma al più n/2 volte, possiamo scrivere, più correttamente, che T(n) = O(n).

Con identica filosofia, la funzione iterativa può essere così scritta:

```
    def Palindromo_ite(A, n)
1       m=0
2       while m<n and A[m]==A[n-1]:
3           m+=1, n-=1
4       if n<=m: return 1
5      return 0
```

Anche il suo costo computazionale è T(n) = O(n) poichè il ciclo while della linea 2 viene eseguito al più *n/2* volte.

8 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dato un array *A* di *n* interi, progettare un algoritmo ricorsivo che restituisce il massimo ed il minimo di *A* in tempo O(n). Verificare tale costo computazionale tramite l'impostazione e la risoluzione di un'equazione di ricorrenza.

Soluzione.

Sapendo di dover produrre un algoritmo ricorsivo, dobbiamo decidere come ridurre il problema ad uno o più sottoproblemi. Le uniche possibilità sensate appaiono le seguenti:

- calcolare ricorsivamente il massimo ed il minimo nei due sottoarray destro e sinistro di dimensione *n/2*, confrontare poi i due massimi ed i due minimi dando in output il massimo e minimo globali;
- calcolare ricorsivamente il massimo ed il minimo del sottoarray di dimensione *n-1* ottenuto eliminando il primo (o l'ultimo) elemento, confrontare poi il massimo ed il minimo ottenuti con l'elemento eliminato dando in output il massimo e minimo globali.

Per comprendere quale dei due approcci sia migliore, proviamo a calcolarne il costo computazionale.
Per quanto riguarda il primo metodo, il tempo di esecuzione su un input di *n* elementi, T(n), è pari al tempo di esecuzione dello stesso algoritmo su ciascuno dei due sottoarray di *n/2* elementi, più il tempo necessario per confrontare i due massimi ed i due minimi; il caso base si ha quando i sottoarray sono di dimensione 1, per cui il massimo ed il minimo coincidono, e l'unica operazione da fare è quella di confrontare tra loro massimi e minimi per dare in output i valori globali. l'equazione di ricorrenza è dunque:

$$\begin{cases} T(n,0) = \Theta(1) \\ T(n) = 2T(n/2) + \Theta(1) \text{ se } n > 1. \end{cases}$$

Risolvendo questa equazione con il metodo iterativo (ma si può fare anche con il metodo dell'albero), si ottiene:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(1) = 2\left(2T\left(\frac{n}{2^2}\right) + \Theta(1)\right) + \Theta(1) = \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2\Theta(1) + \Theta(1) == 2^2\left(2T\left(\frac{n}{2^3}\right) + \Theta(1)\right) + 2\Theta(1) + \Theta(1) = \\ &... = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta(1). \end{aligned}$$

Procediamo fino a quando $n/2^k$ non sia uguale ad 1 e cioè fino a quando $k = log\ n$. In tal caso, sostituendo nell'espressione di T(n) e sfruttando il caso base si ha:

$$T(n) = 2^{\log\ n}\Theta(1) + \Theta(1) \sum_{i=0}^{\log\ n\ -\ 1} 2^i = \Theta(n).$$

Studiamo ora il secondo approccio: il tempo di esecuzione dell'algoritmo *T(n)* su un input di *n* elementi è pari al tempo di esecuzione dello stesso algoritmo su *n - 1* elementi più il tempo per confrontare l'elemento rimasto con il massimo ed il minimo trovati; anche qui il passo base si ha quando *n = 1*. L'equazione di ricorrenza allora diventa:

$$\begin{cases} T(1) = \Theta(1) \\ T(n) = T(n-1) + \Theta(1) \text{ se } n > 1. \end{cases}$$

Anche questa equazione si può risolvere con il metodo iterativo ottenendo:

$$T(n) = T(n-1) + \Theta(1) = T(n-2) + \Theta(1) + \Theta(1) = \cdots = T(n-k) + k\Theta(1).$$

Procediamo fino a quando *n - k = 1* cioè fino a quando *k = n - 1* e sostituiamo nell'equazione: $T(n) = T(1) + (n-1)\Theta(1) = \Theta(n)$ se si tiene conto del caso base.

Deduciamo dai precedenti ragionamenti che entrambi i metodi sono ugualmente corretti ed efficienti.

9 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

L'algoritmo Insertion-Sort può essere espresso come una procedura ricorsiva nel modo seguente:

per ordinare *A[0 . . . n-1]*, si ordina in modo ricorsivo *A[0 . . . n - 2]* e poi si inserisce *A[n-1]* nell'array ordinato *A[0 . . . n - 2].*

Si scriva l'equazione di ricorrenza per il tempo di esecuzione di questa versione ricorsiva di insertion sort.

Soluzione.

Denotiamo con *T(n)* il tempo impiegato nel caso peggiore dalla versione ricorsiva dell'algoritmo Insertion-Sort. Abbiamo che *T(n)* è pari al tempo necessario per ordinare ricorsivamente un array di *n - 1* elementi (che è uguale a *T(n - 1)*) più il tempo per inserire *A[n]* (che nel caso peggiore è $\Theta(n)$). Pertanto la ricorrenza diviene:

$$T(n) = T(n-1) + \Theta(n).$$

Per quanto riguarda il caso base, esso si ha ovviamente per *n = 1* ed, in tal caso, il tempo di esecuzione è $\Theta(1)$.

La soluzione dell'equazione di ricorrenza può essere ottenuta con uno qualunque dei metodi studiati. Qui utilizziamo il metodo iterativo:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) = T(n-2) + \Theta(n-1) + \Theta(n) = \\ &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) = \ldots = T(n-k) + \sum_{i=0}^{k-1} \Theta(n-i). \end{aligned}$$

Raggiungiamo il caso base quando *n - k = 1*, cioè quando *k = n - 1*. In tal caso, l'equazione diventa:

$$T(n) = \Theta(1) + \sum_{i=0}^{n-2} \Theta(n-i) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Si consiglia al lettore di risolvere l'equazione di ricorrenza con gli altri metodi per esercizio.

10 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Dato un array di n numeri reali non nulli, progettare un algoritmo efficiente che posizioni tutti gli elementi negativi prima di tutti gli elementi positivi.

Dell'algoritmo progettato si dia:

a) la descrizione a parole

b) lo pseudocodice

c) il costo computazionale.

Soluzione.

Sia A l'array dato in input. Lo scopo dell'algoritmo che dobbiamo progettare è quello di separare i numeri positivi dai negativi, senza introdurre alcun ordine particolare.
Una funzione simile a quella che partiziona l'array rispetto ad una soglia nel Quicksort potrebbe fare al caso nostro, con l'unica accortezza che la soglia qui deve essere 0. Non avremo difficoltà a gestire elementi dell'array nulli perchè il testo ci assicura che non ce ne sono.
La descrizione a parole è dunque la seguente:

scorri l'array A da sinistra a destra con un indice i e da destra a sinistra con un indice j; ogni volta che i indica un elemento positivo e j un elemento negativo esegui uno scambio; prosegui fino a quando l'array non sia stato completamente visionato.

Lo pseudocodice, del tutto analogo alla funzione di partizionamento del Quicksort, è il seguente:

```
    def PartizionaPosNeg(A):
1       i,j = 0, len(A)-1
2       while i < j:
3           while A[j] > 0 and i <= j : j-=1
4           while A[i] < 0 and i <= j : i+=1
5           if i < j:
6               A[i],A[j]=A[j],A[i]
```

Infine, il costo dell'algoritmo è esattamente lo stesso della funzione a cui ci siamo ispirati, ma ricalcoliamolo qui per completezza:

- la linea 1 ha costo *O(1)*;
- i tre `while` alle linee 2, 3 e 4, complessivamente, fanno in modo che gli elementi dell'array vengano tutti esaminati esattamente una volta e le istruzioni all'interno del ciclo impiegano tempo costante.

Il tempo di esecuzione è indipendente dall'array in input; se ne conclude che il costo è $\Theta(n)$.

Il costo spaziale è anch'esso lineare in *n*, visto che l'algoritmo utilizza l'array di input *A* di lunghezza *n* e due contatori.

Questo esercizio si può anche risolvere osservando che il problema di separare gli elementi positivi dai negativi si può ridurre a quello di ordinare *n* numeri interi nel range [0,1]; si può quindi applicare una modifica dell'algoritmo di Counting Sort, come segue:

```
    def SeparaPosNeg2(A):
1       n=len(A)
2       B=[0]*len(A)
3       neg,pos=0,n-1
4       for i in range(n):
5           if A[i]>0:
6               B[pos]=A[i]; pos-=1
7           else:
8               B[neg]= A[i]; neg+= 1
9       for i in range(n):
10          A[i]=B[i]
```

Si osservi che non abbiamo bisogno del terzo array *C* utilizzato dal Counting Sort, essendo i dati semplicemente degli interi (senza dati satellite). Osserviamo anche che l'algoritmo proposto posiziona gli elementi positivi nell'ordine dato e gli elementi negativi nell'ordine inverso.

Il costo computazionale di questo algoritmo è, ovviamente, lineare in *n*, visto che il suo pseudocodice è costituito dall'istanziazione dell'array ausiliario B di n locazioni e due cicli `for` che eseguono operazioni costanti ad ogni iterazione. Il costo spaziale è anch'esso lineare in *n*, visto che l'algoritmo utilizza, oltre all'array di input *A*, un array ausiliario *B* di lunghezza *n* e tre contatori.

Concludiamo osservando che anche un qualunque algoritmo di ordinamento (per confronti) avrebbe assolto allo scopo, facendo anzi più del richiesto, ma richiedendo anche un costo computazionale di $\Omega(n \log n)$. Se ne deduce che una tale soluzione è senz'altro da scartare.

11 Esercizio.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Dati due insiemi di interi $A = \{a_1, a_2, \dots, a_n\}$ e $B = \{b_1, b_2, \dots, b_m\}$, sia C la loro intersezione, ovvero l'insieme di elementi contenuti sia in A che in B.

Si discutano, confrontandoli, i costi computazionali dei seguenti due approcci al problema:

- applicare un algortmo "naive" che confronta elemento per elemento;
- applicare un algoritmo che prima ordina gli elementi di A e B.

Soluzione.

La soluzione naive è la prima e più semplice che viene in mente: per ogni elemento di A (cioè `for` un certo indice i che va da 1 ad n) si scorre B (cioè `for` un certo altro indice j che va da 1 ad m) per vedere se l'elemento a_i è presente o no in B. Ne segue che il costo computazionale di questo approccio è $O(nm)$. Scriviamo O e non Θ perchè, mentre il ciclo `for i` viene eseguito completamente, il ciclo `for j` si può interrompere appena l'elemento a_i è uguale all'elemento b_j. Pertanto, nel caso migliore, in cui tutti gli elementi di A sono uguali tra loro ed al primo elemento di B, il costo è Θ(n), ma nel caso peggiore, in cui l'intersezione è vuota, il costo è Θ(nm).

Tral'altro, l'analisi del caso migliore fa venire in mente che una trattazione a parte meriterebbe il caso della duplicazione degli elementi; esso si risolve facilmente ricordando che un insieme, per definizione, non ammette ripetizioni al suo interno, e quindi dobbiamo considerare gli a_i tutti distinti tra loro, così come i b_i. Ne discende che il caso migliore presentato non è ammissibile, ma questo non cambia il costo del caso peggiore.
Consideriamo ora il secondo approccio. Si ordinino gli insiemi A e B in modo crescente, e poi si confrontino i due array come segue:

- si pongano un indice *i* all'inizio di *A* e un indice *j* all'inizio di *B*;
- si confronti *A[i]* con *B[j]*:
  - se essi sono uguali, l'elemento si inserisce in *C* e si incrementano sia *i* che *j*,
  - se *A[i]* è minore, si incrementa *i*,
  - se *B[j]* è minore, si incrementa *j*;
- si ripeta dal confronto finchè non si raggiunga la fine di uno dei due array.

Il costo computazionale *T(n, m)* di questo approccio deve tener conto del costo dell'ordinamento e di quello del confronto, e pertanto è pari a

$$\Theta(n \log n) + \Theta(m \log m) + \Theta(max(n, m)).$$

Se assumiamo, senza perdere di generalità, che n $\geq$ m, si ha che

$$T(n, m) = \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

Un'idea alternativa sfrutta l'ipotesi che tutti gli elementi in ciascun insieme siano distinti. Sempre senza perdere di generalità, ponendo n $\geq$ m, possiamo ordinarel'array più piccolo, B, con un algoritmo efficiente (che richiede tempo Θ(m log m)) e poi cercare gli elementi di A in B tramite una ricerca binaria. Ciò può essere fatto in tempo Θ(n log m). In base all'ipotesi n $\geq$ m si ha che:

$$T(n, m) = \Theta(m \log m) + \Theta(n \log m) = \Theta(n \log m).$$

Se potessimo, infine, fare alcune ipotesi sugli elementi di *A* e di *B*, in modo da poter applicare uno degli algoritmi di ordinamento lineari, il costo si abbasserebbe ulteriormente a Θ(n).

12 Esercizio.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Siano dati due array A[1 ... n] e B[1 ... m], composti da n $\geq$ 1 ed m $\geq$ 1 numeri reali, rispettivamente. Gli array sono entrambi ordinati in senso crescente. *A* e *B* non contengono valori duplicati; tuttavia, uno stesso valore potrebbe essere presente una volta in *A* e una volta in *B*.

Progettare un algoritmo iterativo efficiente che stampi i numeri reali che appartengono all'unione dei valori di *A* e di *B*; l'unione va intesa in senso insiemistico, quindi gli eventuali valori presenti in entrambi gli array devono essere stampati solo una volta. Ad esempio, se A = $[1, 3, 4, 6]$ e B = $[2, 3, 4, 7]$,l'algoritmo deve stampare $1, 2, 3, 4, 6, 7$.
Di tale algoritmo:

1. Si dia una descrizione a parole;
2. Si mostri in dettaglio come funziona sui due array dati in esempio sopra;
3. Si scriva lo pseudocodice;
4. Si determini il costo computazionale, in funzione di *n* ed *m*;

Soluzione.

Osserviamo innanzi tutto che viene posta l'attenzione sul problema dei duplicati: sappiamo che A e B contengono elementi tutti diversi tra loro ma possono esistere elementi che stanno sia in A che in B, e questi vanno stampati una volta sola. È chiaro quindi che, prima di stampare un qualunque elemento di A dobbiamo prima verificare se esso sia presente anche in B perchè, in tal caso, possiamo fare a meno di stamparlo (infatti sarà stampato quando sarà il momento di stampare gli elementi di B). Ora, per verificare se ciascuno tra gli *n* elementi di A compaia nell'array B,

impieghiamo tempo $\Theta(nm)$ se per ciascun elemento di A scorriamo B. Ma possiamo osservare che, in questo modo, non stiamo usando l'ipotesi che i due array siano ordinati. Un modo alternativo di procedere è di eseguire, per ciascun elemento di A, una ricerca binaria su B, il che ci porterebbe ad un costo di $O(n \log m)$, ma ancora non stiamo sfruttando tutte le ipotesi perchè così abbiamo usato il fatto che B sia ordinato ma l'ordinamento su A non ci serve a niente. Volendo usare l'ordinamento di entrambi gli array, potremmo proporre un algoritmo simile a quello di fusione del Mergesort, in cui si trascrive solo uno degli elementi uguali. Un tale approccio ci porta ad un costo computazionale di $\Theta(n + m)$, che è anche ottimo visto che nel caso peggiore gli elementi da stampare sono proprio $n + m$.

Fatte queste premesse, possiamo rispondere alle domande:

1. Consideriamo i due array A e B, e progettiamo un algoritmo simile a quello di fusione: consideriamo un indice i che scorre A ed un indice j che scorre B che partono entrambi da 1. Confrontiamo gli elementi A[i] e B[j] e operiamo in modo diverso a seconda dei casi:

   - se A[i] $<$ A[j] si stampa A[i] e si incrementa i
   - se A[i] $>$ A[j] si stampa A[j] e si incrementa j
   - se A[i] = A[j] si stampa A[i] e si incrementano sia i che j.

Appena uno dei due array termina, stampiamo tutti gli elementi dell'altro array. Osserviamo che, a differenza della funzione di fusione del Merge-Sort, non abbiamo qui bisogno di un array ausiliario.

2. Dati i due array A = $[1, 3, 4, 6]$ e B = $[2, 3, 4, 7]$, si pongono i = j = 0. Poichè A[0] = 1 $<$ B[0] = 2 si stampa 1 e si incrementa i, che passa a 1. Si confrontano A[1] = 3 con B[0] = 2 per cui si stampa 2 e si incrementa j, che passa a 1. Si confrontano ora i valori di A[1] = 3 e B[1] = 3; i valori sono uguali per cui se ne stampa solo uno ma entrambi gli indici vengono incrementati, per cui sia i che j passano a 2. Prima di procedere, osserviamo che non possiamo trovare, negli array, altri valori pari a 3 poichè per

ipotesi in A ed in B gli elementi sono tutti distinti. Si confrontano A[2] = 4 e B[2] = 4; essi sono uguali e quindi ne stampiamo uno ed incrementiamo sia i che j passano a 3. A[3] = 6 $<$ B[3] = 7, quindi stampiamo 6 ed incrementiamo i che passa a 4, uscendo fuori dall'array; concludiamo quindi stampando gli elementi nella parte rimanente dell'array B, cioè 7.

3. Questo è un possibile codice:

```
def StampaUnione(A, B):
    n,m = len(A),len(B)
    i=j=1
    while i < n and j < m:
        if A[i] < B[j]:
            print(A[i]); i+=1
        elif A[i] > B[j]:
            print(B[j]); j+=1
        else:
            print(A[i]); i+=1; j+=1
    while i < n:  #  stampa la parte finale del vett. A non ancora terminato
        print(A[i]); i+=1
    while j < m:  #  stampa la parte finale del vett. B non ancora terminato
        print(B[j]); j+=1
```

4. Il costo di questo algoritmo $\Theta(n + m)$ in quanto sostanzialmente si effettua una scansione di entrambi gli array esaminando ciascun elemento una e una sola volta in tempo $\Theta(1)$.

13 Esercizio.

6. Il problema dell'ordinamento

Dato un array A di n interi non negativi distinti, si vuole determinare se esistono almeno tre numeri consecutivi di valore inferiore a 100.
Ad esempio, se A = $[101, 5, 9, 31, 33, 10, 100, 4, 8, 32, 500, 11, 99]$, gli elementi 8, 9 e 10 così come gli elementi 31, 32 e 33 rispettano la proprietà mentre 99, 100 e 101 no.
Progettare un algoritmo che, dato A, in tempo $\Theta(n)$ restituisce il valore dell'elemento centrale della terna se questa è presente, -1 altrimenti. Se esistono più terne allora bisogna restituire l'elemento centrale di valore massimo (nell'esempio sopra, l'algoritmo dovrebbe restituire 32).
Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato;

b) si giustifichi il costo computazionale.

Soluzione.

Osserviamo preliminarmente che l'algoritmo naïf che consiste nello scorrere l'array per ciascun elemento alla ricerca dei due elementi che possano far parte della terna ha un tempo quadratico, mentre ordinare l'array e poi scorrerlo una sola volta alla ricerca di una eventuale terna richiederebbe tempo $\Omega(n \log n)$ (il costo esatto dipende dall'algoritmo di ordinamento usato). In entrambi i casi si arriva ad un costo troppo alto.
Si propongono qui vari modi di procedere, tutti basati sul fatto che la soluzione va cercata tra i valori compresi tra 0 e 99.

Una prima idea consiste nell'utilizzare una variante dell'algoritmo di ordinamento per conteggio, con un array di lavoro C di dimensione 100 dove, in posizione C[i], viene inserito il numero di occorrenze in A dell'intero i, **solo se** $i < 100$. È da notare che eseguire una semplice chiamata dell'algoritmo di Counting Sort sull'intero array A NON garantisce il costo

Θ(n) richiesto, che è lineare nella dimensione dell'array E nell'ampiezza dell'intervallo dei dati.
Con una singola scansione dell'array A si può riempire l'array C, e con al più una singola scansione dell'array C si può scoprire se è presente almeno una terna.
Si noti che l'elemento centrale di una terna è un intero i tra 1 e 98 tale che C[i$-$1], C[i] e C[i+1] hanno valore diverso da zero e che l'elemento cercato è il primo elemento centrale di una terna scorrendo l'array C da destra, mentre se si scorre C da sinistra bisognerà tenere traccia dell'elemento centrale maggiore via via trovato.
Al termine viene restituito -1 se nessuna terna è stata individuata o il valore centrale dell'ultima terna trovata.

Un'altra possibilità consiste nell'applicare un algoritmo che antepone tutti gli elementi $\leq$ 100 agli altri (possiamo quindi scegliere 100 come soglia del classico algoritmo di partizionamento); si può poi procedere all'ordinamento della parte sinistra dell'array in tempo costante (poiché sono tutti distinti non saranno più di 100) ed alla ricerca della terna come sopra.

Un'ulteriore idea è quella di scorrere l'array A e, per ogni elemento A[i] $<$ 100, cercare nell'array il precedente ed il successivo di A[i]. Anche qui viene garantito il costo lineare perché si scorre l'array A al più 100 volte, quindi comunque Θ(n).

Segue un possibile codice Python della prima idea proposta.

```
def terna(A):
    C = [0] * 100
    for i in range (len(A)):
        if A[i] < 100:    C[A[i]] += 1
    t = -1
    for i in range(1, 99):
        if C[i - 1] > 0 and C[i] > 0 and C[i + 1] > 0:    t = i
    return t
```

Il costo è ottenuto sommando il contributo del primo *for* (che è $\Theta(n)$ a quello del secondo *for* (che è $O(1)$). Ne segue un costo complessivo di $\Theta(n)$.

14 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Si consideri il codice del seguente algoritmo di ordinamento. dove la funzione viene richiamata la prima volta con *i=1* e *j=len(A)-1*.

```
    def  UnAltroSort (A, i, j):
1       if A[i] > A[j]:
2           A[i],A[j]= A[j],A[i]
3       if i+1 >= j: return
4       k=(j-i+1)//3
5       UnAltroSort(A, i, j-k) # si ordinano i primi 2/3 dell'array
6       UnAltroSort(A, i+k, j) # si ordinano gli ultimi 2/3 dell'array
7       UnAltroSort(A, i, j-k) # si ordinano di nuovo i primi 2/3 dell'array.
```

1. si scriva l'equazione di ricorrenza che descrive il costo computazionale della funzione nel caso peggiore, dandone giustificazione;
2. si risolva l'equazione trovata usando il teorema principale ed un altro metodo e, per ciascuno, si mostri il procedimento usato;
3. si confronti il tempo di esecuzione del caso peggiore di `UnAltroSort` con quello dell'insertionSort e del MergeSort, facendo le opportune considerazioni sull'efficienza dell'algoritmo proposto.

Soluzione.

1. Innanzi tutto dobbiamo individuare il parametro dell'equazione di ricorrenza. In questo ci aiutano i commenti, i quali ci suggeriscono che le chiamate ricorsive vengono effettuate su un array di lunghezza pari ai 2/3 dell'array originario; pertanto, è naturale assumere che il parametro sia proprio la lunghezza dell'array; usando i nomi delle variabili presenti nella funzione, tale parametro si può esprimere come $n = j - i + 1$. Questo è in accordo con le linee 5, 6 e 7, dove le lunghezze dei tre sottoarray sui quali viene richiamata la funzione sono in tutti e tre i casi all'incirca $2/3n$.

Si ricordi ora che un'equazione di ricorrenza si compone della parte relativa al caso generale e quella relativa al caso base.
Cominciamo quindi ad individuare la linea di codice relativa al caso base. Essa è ovviamente, la riga 3, e quindi interessa il caso in cui i + 1 $\geq$ j. Poichè *i* e *j* rappresentano rispettivamente l'inizio e la fine dell'array in considerazione, siamo in un caso base tutte le volte che l'elemento successivo a quello di indice *i* si trova in corrispondenza di *j* oppure alla sua destra, quindi il caso base si ottiene quando n $\leq$ 2 ed, in tal caso, il costo è pari a $\Theta(1)$ poichè vengono eseguite le linee da 1 a 3, tutte operazioni di costo costante.
Nel caso generale, invece, il costo è dato dal contributo $\Theta(1)$ proveniente dalle linee 1-4 e dal contributo delle tre linee 5-7, ciascuna delle quali ha costo *T(2/3 n)*, per cui:

$$\begin{cases} T(n) = \Theta(1) + 3T(2n/3) \text{ se } n > 2 \\ T(1) = T(2) = \Theta(1). \end{cases}$$

2. Questa equazione di ricorrenza si può risolvere tramite teorema principale, tramite metodo iterativo o metodo dell'albero, e la soluzione è $T(n) = \Theta(n^{\log_{3/2} 3})$.

Poichè la soluzione non presenta particolari difficoltà, omettiamo qui i dettagli.

3. Confrontiamo ora il costo della funzione data con quella dell'InsertonSort, $\Theta(n^2)$, e del MergeSort, $\Theta(n\log n)$. Per fare ciò , dobbiamo farci un'idea del valore di $\log_{3/2} 3$. Osserviamo che la funzione logaritmica è una funzione monotona crescente, e quindi:

$$1 = \log_{3/2} 3/2 < 2 = \log_{3/2} 9/4 < \log_{3/2} 3.$$

Pertanto $\log_{3/2} 3$ è un valore strettamente maggiore di 2. Ne consegue che il costo di `UnAltroSort` è peggiore sia di quello di MergeSort che di quello di InsertionSort.

15 Esercizio.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Si consideri la seguente modifica dell'algoritmo di Merge Sort, che prende come parametri un array V di interi e due indici `primo` e `ultimo` (che alla prima chiamata valgono 0 ed len(V) − 1 rispettivamente):

```
    def  MergeSortModificato(V, primo, ultimo):
1       n=ultimo-primo+1
2       if n <= 1: return
3       medio1=primo+n//3
4       medio2=medio1+n//3
5       MergeSortModificato(V, primo, medio1)
6       MergeSortModificato(V, medio1 + 1, medio2)
7       MergeSortModificato(V, medio2 + 1, ultimo)
8       Fondi(V, primo, medio1, medio2, ultimo)
9       return
```

Tenendo conto che il costo computazione della funzione `Fondi` è comunque $\Theta(n)$, si ricavi l'equazione di ricorrenza che esprime il costo computazionale della funzione `MergeSortModificato`, specificando i contributi delle varie istruzioni, e si risolva l'equazione di ricorrenza trovata utilizzando il metodo iterativo.

Infine, si discuta se sia possibile includere questo procedimento tra gli algoritmi di ordinamento efficienti, e se si possa ulteriormente generalizzare l'idea di suddividere l'array di partenza in un numero qualsiasi k di sottoarray.

Soluzione.

Questo pseudocodice è, ovviamente, estremamente simile al Merge Sort, salvo il fatto che l'array viene diviso ricorsivamente in tre parti anzichè in due. Pertanto, l'equazione di ricorrenza è, in questo caso:

$$\begin{cases} T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n) \text{ se } n > 1 \\ T(1) = \Theta(1) \end{cases}$$

la cui soluzione, che si può determitare tramite uno qualunque dei metodi studiati, è $\Theta(n \log n)$. È quindi evidente che questo algoritmo può tranquillamente rientrare tra gli ordinamenti efficienti.

Affrontiamo ora l'ultimo punto, che è forse il più interessante: cosa succede se, invece di dividere l'array in tre parti, generalizzassimo il ragionamento e lo dividessimo in *k* parti? Dobbiamo considerare che, mentre la funzione `Fondi` applicata a tre sotto-array ha comunque un costo lineare (perchè ad ogni passo selezioniamo il massimo tra 3 oggetti anzichè tra 2), quando generalizziamo a *k* sotto-array, questo non è più vero e la selezione di ciascun minimo costerà $\Theta(k)$, per cui l'intera funzione `Fondi` avrà un costo di $\Theta(kn)$, e questo va sostituito nell'equazione di ricorrenza al posto di $\Theta(n)$ dando luogo alla seguente soluzione:

$$T(n) = kT(n/k) + \Theta(kn) = k[kT(n/k^2) + \Theta(k\frac{n}{k})] = k^i T(\frac{n}{k^i}) + i\Theta(kn) =$$

... procedendo finchè $\frac{n}{k^i} = 1$ cioè finchè $i = \log_k n$ ...

$$= k^{\log_k n}\Theta(1) + \log_k n O(kn) = \Theta(n) + O(kn \log_k 2 \log_2 n).$$

Poichè non è restrittivo assumere che k sia almeno 2, il valore di $log_k 2$ è sempre limitato da una costante inferiore ad 1 e quindi la soluzione è $\Theta(n\ k \log n)$, che è strettamente superiore a $\Theta(n \log n)$ se *k* non è costante. Segue che questa ulteriore generalizzazione non si può annoverare tra gli algoritmi di ordinamento ottimi.

16 Esercizio.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Sia dato un array *A* contenente *n* interi distinti e ordinati in modo crescente.

Progettare un algoritmo che, in tempo *O(log n)*, individui la posizione più a sinistra nell'array per cui si ha $A[i] \neq i$,l'algoritmo restituisce -1 se una tale posizione non esiste.

Ad esempio, per A = $[0, 1, 2, 3, 4]$, l'algoritmo deve restituire -1, per A = $[0, 5, 6, 20, 30]$, la risposta deve essere 1 e, per A = $[-3, 1, 2, 3, 6]$, la risposta deve essere 0.

Soluzione.

La prima cosa che pensiamo è che, per ottenere un tempo *O(log n)*, possiamo ricorrere ad una versione modificata della ricerca binaria. Facendo qualche piccolo esempio, però , non è immediatamente chiaro capire come. Infatti, se consideriamo l'elemento centrale dell'array, nel caso in cui questo sia uguale al proprio indice, si può dover andare a sinistra o a destra secondo una regola che non è subito chiara. Tuttavia, quello che è evidente che non possiamo progettare un algoritmo che vada prima a sinistra e poi (se serve) anche a destra, perchè altrimenti perdiamo il costo logaritmico e questo non è accettato.

Notiamo che, se i valori dell'array fossero non negativi, le cose sarebbero molto più facili, perchè, nel caso in cui *A[m]* sia uguale ad *m* (come *m* indice dell'elemento di mezzo), dovrebbe necessariamente accadere che l'intera parte destra gode della stessa proprietà, perchè l'unico modo di sistemare *m+1* elementi interi, non negativi e distinti in *m+1* posizioni consiste nell'usare valori consecutivi da 0 ad *m*. Fatta questa osservazione, ci rendiamo conto che è facile capire se gli elementi del nostro array sono tutti non negativi oppure no: basta guardare il primo elemento. Non solo: il primo elemento ci dice molto di più! Infatti, se è un valore non nullo, certamente non coincide con il suo indice (0) e quindi abbiamo già trovato

l'indice cercato (è certamente il più a sinistra), se è proprio 0 sappiamo che non ci sono elementi negativi e quindi possiamo passare a studiare l'elemento centrale. Ciò vuol dire che, se ci accingiamo a controllare l'elemento centrale, certamente *A[0]=0*.

Per prima cosa, allora, verifichiamo se già il primo elemento coincida con la sua posizione, in caso contrario restituiamo quella posizione, cioè 0.

Altrimenti, confrontiamo l'elemento centrale dell'intervallo con il suo indice *m*; se si ha *A[i]=m* allora l'elemento mancante, se c'è, deve essere necessariamente nell'intervallo di destra (perchè, come già osservato, le posizioni alla sinistra sono occupate dai numeri che vanno da 0 a *m*); se, al contrario, $A[m] \neq m$, questo elemento potrebbe essere la soluzione giusta, ma potrebbe anche essercene una più a sinistra, e quindi dobbiamo cercare nell'intervallo a sinistra.

Verrà restituito -1 solo se si arriva all'intervallo "vuoto" (vale a dire l'intervallo con indici iniziale e finale *i* e *j* con *i>j*.

Una volta fatte tutte queste considerazioni, è immediato proporre una possibile implementazione dell'algoritmo (ad esempio in Python) che utilizzi una versione ricorsiva della ricerca binaria modificata. La prima volta la sotto-funzione deve essere invocata sull'intero array, vale a dire:
`es2(A, 0, len(A)-1).`

```
def es2(A):
    if A[0]!=0:  return 0
    return es2R(A, 0, len(A)-1).


def es2R(A, i, j):
    if i>j:
        return -1
    if A[i] != i:
        return i
    m = (i+j)//2
    if A[m] != m:
        return es2R(A, m+1, j)
    else:
        return es2R(A, i, m)
```

Per verificare che il costo computazionale sia effettivamente logaritmico, basta ricavare la ricorrenza associata al costo computazionale dell'algoritmo, che è quella tipica della ricerca binaria, vale a dire $T(n) \leq T(\frac{n}{2}) + \Theta(1)$ per $n \geq 1$, $T(n)=\Theta(1)$ altrimenti.

Questa ricorrenza risolta, ad esempio con il teorema principale ha come soluzione $O(\log n)$.

17 Esercizio.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Dato un heap *H* di *n* elementi, descrivere a parole l'algoritmo che cancella un prefissato nodo *i* e riaggiusta l'heap risultante di *n - 1* elementi.

Valutare il costo computazionale dell'algoritmo presentato.

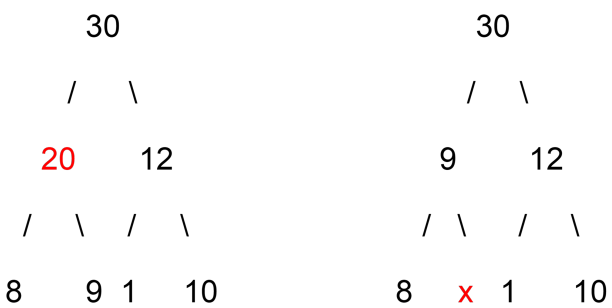
Soluzione.

Il problema si scompone in due parti: la prima consiste nel cercare l'elemento da eliminare e l'altra nel cancellarlo e ripristinare l'heap.

l'elemento *i* da eliminare può essere inteso come l'elemento di posizione *i* oppure l'elemento di chiave *i*. Nel primo caso l'individuazione dell'elemento richiede tempo costante (l'elemento è *H[i]*), nel secondo bisogna effettuare una ricerca, che può richiedere anche tempo lineare in *n*, cioè *O(n)*.

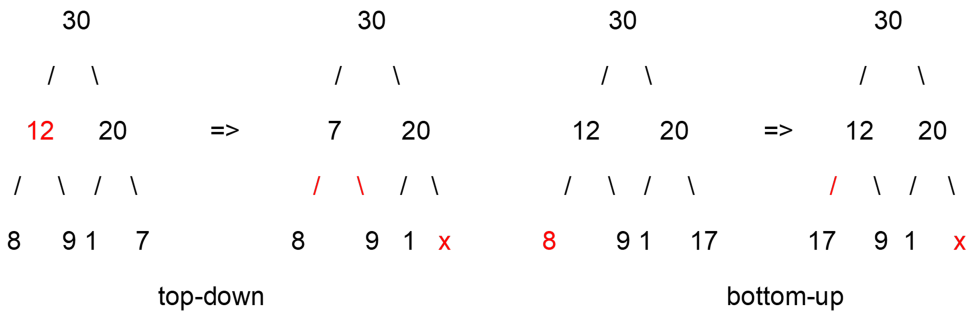
Una volta noto l'elemento, sia esso *H[x]*, è possibile eliminarlo apparentemente in molti modi, alcuni dei quali sono sintetizzati qui:

- si può portare verso le foglie l'elemento da cancellare scambiandolo con il maggiore dei suoi figli; una volta che l'elemento ha raggiunto le foglie, si elimina semplicemente; questo algoritmo è errato perchè un heap è un albero binario completo o quasi completo e questo approccio può far perdere questa proprietà; si veda, ad esempio, nell'esempio qui sotto, cosa succede quando si tenta di eliminare il nodo di chiave 20 con questo approccio:



- si può shiftare ogni elemento *H[x+1], H[x+2], . . . , H[n]* di una posizione verso sinistra e poi riaggiustare l'heap, ma lo shift a sinistra provoca un disallineamento dei figli rispetto ai padri per cui nessun nodo è garantito mantenere la proprietà dell'heap, che - per essere aggiustato - dovrebbe essere interamente ricostruito; si veda l'esempio qui sotto:

- si può scambiare *H[x]* con *H[n]*, cioè con la foglia più a sinistra, eliminare tale foglia e riaggiustare l'heap risultante. Per fare ciò , bisogna confrontare il nuovo *H[x]* con il maggiore dei suoi figli; se *H[x]* risulta minore, si può applicare la funzione di aggiustamento dell'heap top-down; se *H[x]* sembra in posizione corretta rispetto ai suoi figli, non è ancora detto che l'heap sia corretto, bisogna infatti confrontarlo con il padre, se questo è minore di *H[x]* si deve procedere all'aggiustamento bottom-up:



Se escludiamo la ricerca del nodo, l'algoritmo di cancellazione e riaggiustamento ha una complessità pari al massimo tra la complessità dell'aggiustamento top-down e di quello bottom-up. Poichè entrambi si possono eseguire in tempo *O(log n)*, ne consegue che *O(log n)* è proprio il costo dell'algoritmo presentato.

18 Esercizio.

Riferimento ai capitoli: 7. Strutture dati fondamentali

Siano date k liste ordinate in cui sono memorizzati globalmente n elementi. Descrivere un algoritmo con tempo O(n log k) per fondere le k liste ordinate in un'unica lista ordinata.

Soluzione.

Si consideri, innanzi tutto, il classico algoritmo di fusione applicato a due liste:

> al generico passo *i*, sono già stati sistemati nella lista finale *i* elementi, e vi sono due puntatori alle teste (contenenti gli elementi più piccoli) delle due liste; si individua quale tra i due elementi puntati sia il minore, e questo viene inserito nella lista risultante in coda. Si passa quindi al passo *i+1*.

Il numero di passi complessivi àovviamente pari ad *n*, cioè al numero complessivo di elementi.

È naturale cercare di generalizzare questo algoritmo a *k* liste ordinate, purchè si sia in grado di calcolare il minimo fra *k* elementi. Banalmente, questo si può fare in tempo $\Theta(k)$, producendo quindi un algoritmo di fusione di *k* liste di costo $\Theta(nk)$.

Ma il tempo richiesto dal testo deve essere più basso, quindi dobbiamo trovare il modo di lavorare più efficientemente. Posto che non sembra possibile ridurre il numero di iterazioni, che dovrà comunque rimanere *n*, non resta che ragionare sulla ricerca del minimo tra *k* elementi.

Richiamiamo alla memoria il fatto che una struttura dati in grado di restituire efficientemente il valore minimo, tra quelli che contiene, è il min-heap. Allora, al generico passo *i*, i valori delle *k* teste delle *k* liste ordinate sono memorizzati in un min-heap, costruito in tempo $\Theta(k)$, da cui si può estrarre

il minimo in tempo *O(log k)*; una volta eliminato tale valore dalla struttura ed aggiornato il puntatore relativo alla testa della lista corrispondente, dovremo inserire nel min-heap il nuovo valore minimo, ed anche questo impiega un tempo *O(log k)*.

Iterando questo procedimento per tutti gli *n* passi, si giunge ad un costo di *O(k+n log k) = (n log k)*, che è quella richiesta.

19 Esercizio.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

Sia dato un albero binario completo con n nodi, radicato al nodo puntato dal puntatore r. Calcolare il costo computazionale della seguente funzione, in funzione di n:

```
    def Funzione( r):
1       if !r.fs and !r.fd: return r
2       r1=r;
3       while r1.fs: r1=r1.fs
4       fogliasx=r1.dato
5       r1=r
6       while r1.fd: r1=r1.fd
7       fogliadx=r1.dato
8       if  fogliasx < fogliadx: return Funzione(r.fs)
9       return Funzione(r->fd)
```

Soluzione.

La funzione è ricorsiva e pertanto dobbiamo trovare un'equazione di ricorrenza in funzione di n; il costo si ottiene sommando i contributi delle varie istruzioni:

- la linea 1 rappresenta il caso base, che costa $\Theta(1)$;
- le linee 2, 4, 5 e 7 costano $\Theta(1)$;
- i cicli while delle linee 3 e 6 vengono ripetuti un certo numero di volte indefinto tra 1 e l'altezza dell'albero, infatti il primo ciclo si interrompe quando il nodo non ha figlio sinistro ed il secondo ciclo quando il nodo non ha figlio destro;

- le linee 8 e 9 sono chiamate ricorsive e quindi il loro costo si dovrà scrivere come T(?) con un opportuno valore tra le parentesi, ma quale?

Se indichiamo con *k* ed *n -1 - k* il numero dei nodi dei sottoalberi radicati ai figli sinistro e destro della radice, otteniamo:

$$T(n) = \Theta(1) + v_1\,\Theta(1) + v_2\,\Theta(1) + max(T(k), T(n-k))$$

dove v_1 e v_2 sono il numero di volte in cui vengono ripetuti i cicli delle linee 5 e 9.

Arrivati qui, non sappiamo come procedere, se non sostituendo a v_1 e v_2l'altezza dell'albero, che ne è una limitazione superiore.l'altezza ha un valore indefinito tra *log n* ed *n*, e questo significa che dobbiamo anche qui sostituire ad *h* la sua limitazione superiore, cioà*n*. Viste tutte queste approssimazioni che siamo costretti a fare, dovrebbe venire il dubbio di aver tralasciato qualcosa. Rileggendo con attenzione il testo, si osservi chel'albero in input àbinario e ***completo***. Questa ipotesi, che abbiamo tralasciato finora, risolove numerosi problemi, infatti:

- tutti i nodi privi di figlio sinistro (ed anche di figlio destro) sono ad altezza *log n*, pertanto i due cicli alle linee 5 e 9 vengono ripetuti esattamente *log n* volte ciascuno;
- diventa facile calcolare *max(T(k), T(n - k))*: tanto il sottoalbero sinistro che quello destro contengono esattamente *(n-1)/2* nodi ciascuno.

l'equazione di ricorrenza diventa allora:

$$\begin{cases} T(n) = \Theta(1) + \Theta(\log n) + T\left(\frac{n}{2}\right) = \Theta(\log n) + T\left(\frac{n}{2}\right) \text{ se } n > 1 \\ T(1) = \Theta(1) \end{cases}$$

Risolvendo per iterazione si ha:

$$T(n) = \Theta(\log n) + T\left(\frac{n}{2}\right) = \Theta(\log n) + (\Theta(\log \frac{n}{2}) + T(\frac{n}{2^2})) = ... =$$

$$= \sum_{i=0}^{k-1} \Theta(\log \frac{n}{2^i}) + T(\frac{n}{2^i}).$$

Si procede fino a quando $\frac{n}{2^k} = 1$ e cioè fino a quando $k = \log n$, ottenendo:

$$T(n) = \sum_{i=0}^{\log n-1} \Theta(\log \frac{n}{2^i}) + T(1) = \Theta(\sum_{i=0}^{\log n-1} (\log \frac{n}{2^i})) + T(1) =$$

$$= \Theta(\sum_{i=0}^{\log n-1} (\log n - \log 2^i)) + T(1) = \Theta(\sum_{i=0}^{\log n-1} (\log n - i)) + T(1) =$$

$$= \Theta(\log n \sum_{i=0}^{\log n-1} 1 - \sum_{i=0}^{\log n-1} i) + T(1) = \Theta(\log^2 n + \frac{\log n \log(n-1)}{2}) + T(1) =$$

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

20 Esercizio.

Riferimento ai capitoli: 7. Strutture dati fondamentali

Si vuole simulare una coda tramite due pile. In particolare, si possono usare -senza dettagliarle- le seguenti funzioni:

`p=inserisciPila()` che prende in input il puntatore ad una pila ed un valore x, lo inserisce nella pila e restituisce il puntatore alla pila;
`p,x=estraiPila(p)` che prende in input il puntatore ad una pila ne estrae un elemento e restituisce il puntatore alla pila e l'elemento estratto;
`test_di_pila_vuota(p)` che prende in input il puntatore ad una pila e restituisce 0 o 1 a seconda che la pila sia vuota o meno.

Si descrivano a parole le operazioni di `inserisciCoda()` ed `estraiCoda()` con le due pile; si scriva lo pseudocodice delle due funzioni e si valuti il loro costo computazionale.

Soluzione.

Preliminarmente, chiariamo cosa viene richiesto dall'esercizio. Abbiamo due strutture dati di tipo pila (presumibilmente una - ad esempio la prima - usata per memorizzare i dati ed una - la seconda - di appoggio) con le quali dobbiamo simulare una coda. Questo significa, in particolare, che se vogliamo eseguire l'operazione `estraiCoda` dovremo estrarre l'elemento che ànella struttura da più tempo, mentre le pile ci offrono la possibilità di estrarre quello che àstato inserito per ultimo; analogamente, se vogliamo eseguire l'operazione `inserisciCoda`, dovremo inserire l'elemento immediatamente dopo quello che àstato inserito al passo precedente.

Abbiamo in realtà più di un modo per procedere. Consideriamo quello più semplice, in cui cioàle due funzioni `inserisciCoda` (per la coda virtuale) e `inserisciPila` (per la pila) coincidono. Ciò è perfettamente ragionevole, visto che in entrambe le strutture dati gli elementi da inserire vengono accodati nella prima posizione libera.

Perciò , possiamo già scrivere:

```
def inserisciCoda(x,p1):
    return inserimentoPila(x,p1)
```

Viene dunque passato l'elemento x da inserire e il puntatore alla pila di memorizzazione e viene restituito il puntatore alla pila di memorizzazione (che potrebbe essere cambiato nel caso la pila era inizialmente vuota) Il costo computazionale di questa operazione è, ovviamente, *O(1)*.

Le cose si complicano leggermente per l'operazione di `estraiCoda`, infatti per estrarre dalla pila l'elemento che vi giace da più tempo usando solo le operazioni su di essa permesse (`estraiPila` e `inserisciPila`) dobbiamo estrarre uno ad uno tutti gli elementi, appoggiandoli temporaneamente nella seconda pila, così da ribaltarne l'ordine; a questo punto possiamo estrarre dalla seconda pila l'ultimo elemento (corrispondente a quello che àstato inserito nella prima pila per primo) e poi ripristinare la situazione.

Seguendo questa filosofia, la funzione può essere la seguente:

```
def estraiCoda(p1):
    while test_di_pila_vuota(p1)==FALSE:
         p1, x = estraiPila(p1)
        p2 = inserisciPila(x,p2)
    p2, aux = estraiPila(p2)
    while  test_di_pila_vuota(p2)==FALSE:
        p2, x = estraiPila(p2)
         p1 = inserisciPila(x,p1)
    return p1, aux.key
```

Tramite la funzione precedente, svuotiamo la prima pila un elemento per volta e la riversiamo sulla seconda, per poi - tolto l'elemento da estrarre - svuotare la seconda riversandola di nuovo nella prima; il risultato ottenuto

è che la prima pila conterrà gli stessi elementi di prima nello stesso ordine, salvo il primo che sarà stato estratto. La funzione prende in input il puntatore alla pila di memorizzazione e restituisce il valore estratto dalla coda e il puntatore alla pila di memorizzazione (che potrebbe essere cambiato nel caso in cui la pila conteneva un unico elemento).
Detto *n* il numero di elementi che si trovano al momento nella pila, il costo computazionale di questa funzione àovviamente $\Theta(1)n+ \Theta(1)+ \Theta(1)(n-1)+ \Theta(1)= \Theta(n)$.

Possiamo osservare che il numero di operazioni eseguite potrebbe essere diminuito nel caso in cui eseguiamo una sequenza di operazioni `estraiCoda`; infatti, supponiamo di dover estrarre due elementi uno dopo l'altro: per la prima estrazione dovremo riversare la prima pila nella seconda (*n* operazioni `estraiPila` degli elementi restituiti da altrettante operazioni `inserisciPila`) e, una volta estratto l'elemento, riversare nuovamente la seconda nella prima (*n-1* operazioni `inserisciPila` degli elementi restituiti da altrettante operazioni `estraiPila`), per la seconda estrazione dovremo fare tutto da capo, quindi riversare la prima pila nella seconda (*n-1* operazioni `estraiPila` degli elementi restituiti da altrettante operazioni `inserisciPila`) e, una volta estratto l'elemento, riversare nuovamente la seconda nella prima (*n-2* operazioni `estraiPila` dell'elemento restituito da altrettante operazioni `inserisciPila`). Se però potessimo essere in grado di capire che abbiamo due operazioni di tipo `estraiCoda` una dopo l'altra, potremmo fare a meno di riversare la seconda pila nella prima, e quindi anche di riversare dopo, ancora una volta, la prima nella seconda, dimezzando di fatto il numero di operazioni. Si può "reagire" differentemente a seconda della sequenza di richieste di `inserisciCoda` e `estraiCoda` operando sulla funzione che si occupa del menu delle scelte. Per brevità, omettiamo dettagli ulteriori su questo punto.

Vogliamo infine sottolineare come la soluzione proposta non sia l'unica: se decidiamo di non riportare indietro i dati dalla seconda alla prima pila, pos-

siamo fare in modo che le due funzioni `inserisciCoda` e `estraiCoda` procedano in modo differente a seconda che i dati si trovino nella prima o nella seconda pila.

Più precisamente, la funzione `inserisciCoda` è identica alla `inserisciPila` (come prima) se i dati sono nella pila 1 oppure se entrambe le pile sono vuote, mentre se i dati sono nella pila 2 allora essi saranno nell'ordine inverso, quindi dobbiamo prima ribaltarli nella pila 1 e poi aggiungere l'elemento. Analogamente, la funzione `estraiCoda` è identica alla `estraiPila` se i dati sono nella pila 2, ma vanno prima ribaltati nella pila 2 se essi sono nella pila 1. Ovviamente, in questo modo, entrambe le funzioni avranno un costo computazionale di *O(n)* mal'approccio non è peggiore del primo poichè la differenza è che mentre nella prima proposta abbiamo semplificato la `inserisciCoda` lasciando alla `estraiCoda` un doppio riversamento (dalla 1 alla 2 e poi indietro dalla 2 alla 1), qui abbiamo suddiviso i due riversamenti tra le due funzioni.

21 Esercizio.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

L'albero di Kalkin-Wilf è un albero binario (infinito) che nei suoi nodi contiene ciascun numero razionale positivo esattamente una volta. È così definito:

1) se un nodo ha chiave *a/b*:
   - il suo figlio sinistro ha chiave *a/(a + b)*
   - il suo figlio destro ha chiave *(a + b)/b*
2) la radice ha chiave 1/1.

Progettare una funzione ricorsiva che, dato un array di *n* elementi (con indici da *0* a *n-1*) considerato come un albero binario completo o quasi completo gestito con la notazione posizionale, lo riempia con i valori dei corrispondenti nodi dell'albero Kalkin-Wilf.

Soluzione.

Innanzi tutto, osserviamo che, mentre nella definizione originale,l'albero di Kalkin-Wilf è infinito e quindi non presenta alcun caso base per la terminazione della costruzione, il nostro output deve essere un albero binario con n nodi.
È allora facile scrivere una funzione ricorsiva che riempiel'array dato a partire dalla posizione di indice 0, che corrisponde alla radice dell'albero.

```
def crea_albero(V, n, ind, num, den):
    if (ind >= n): return
    V[ind] = num/den
    crea_albero(V, n, 2*ind+1, num, num + den)
    crea_albero(V, n, 2*(ind+1), num + den, den)
    return
```

Questa funzione viene richiamata la prima volta con i parametri ind = 0 e num = den = 1.

Calcoliamo, infine, il costo computazione. Possiamo scegliere se utilizzare come parametro il numero di nodi nel sottoalbero o la distanza dalle foglie del nodo di posizione ind. Se utilizziamo la prima opzione, e chiamiamo *h* la distanza dalle foglie, l'equazione di ricorrenza è:

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

Il caso base è rappresentato dalla chiamata sull'albero vuoto (con ind > n). È immediato verificare che la soluzione di questa equazione di ricorrenza è $\Theta(n)$.

22 Esercizio.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

Lo *sblanciamento* di un nodo di un albero binario è il modulo della differenza tra il numero di nodi del suo sottoalbero sinistro ed il numero di nodi del suo sottoalbero destro. Il massimo tra gli sbilanciamenti dei nodi di un albero àlo sbilanciamento dell'albero (nel caso di albero vuoto il suo sbilanciamento à0). Dato un albero binario di cui si conosce il puntatore alla radice, bisogna progettare una funzione ricorsiva che calcoli lo sbilanciamento dell'albero.

Soluzione.

Cominciamo con l'osservare che, affinchè ciascun nodo calcoli il modulo della differenza tra numero di nodi nel suo sottoalbero sinistro e numero di nodi nel suo sottoalbero destro, ànecessario che esso riceva opportune informazioni da entrambi i suoi figli e poi effettui il calcolo. Per questo, la funzione che dobbiamo scrivere dovrà seguire la filosofia della visita in post-ordine. Considera la seguente funzione ricorsiva:

```
def calcola_sbilanciamentoR(p):
    if !p: return 0, 0
    nodi_sin,sbi_sin=calcola_sbilanciamentoR(p.sin)
    nodi_des,sbi_des=calcola_sbilanciamentoR(p.des)
    sbi= abs(nodi_sin - nodi_des)
    return nodi_sin + nodi_des +1, max(sbi_sin, sbi_des, abs(nodi_sin - nodi_des))
```

La funzione appena scritta, invocata su un certo nodo *p*, riceve ricorsivamente il numero di nodi nei suoi due sottoalberi dai suoi figli e lo sbilanciamento massimo per ciascuno degli alberi dei suoi figli, da queste informazioni calcola il numero di nodi presenti nel sottolabero di cui àradice e lo sbilanciamento massimo tra i nodi del sottoalbero di cui àradice. Trasmette quindi questi due valori al nodo padre. Ovviamente la funzione

richiesta dall'esercizio si ottiene banalmente dalla procedura ricorsiva con un costo addittivo di $\Theta(n)$.:

```
def calcola_sbilanciamento(p):
    nodi, sbi = calcola_sbilanciamentoR(p)
    return sbi
```

Il costo computazionale di `calcola_sbilanciamentoR` è, come àovvio, quello della visita di un albero, e l'equazione di ricorrenza relativa allo pseudocodice è:

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

che si può risolvere con il metodo di sostituzione (v. Dispense di teoria) dando come soluzione $\Theta(n)$.

23 Esercizio.

Riferimento ai capitoli: 8. Dizionari

Siano dati due alberi binari di ricerca: B_1 con n_1 nodi ed altezza h_1, e B_2 con n_2 nodi ed altezza h_2. Assumendo che tutti gli elementi in B_1 siano minori di quelli in B_2, descrivere un algoritmo che fonda gli alberi B_1 e B_2 in un unico albero binario di ricerca B di $n_1 + n_2$ nodi.

Determinare l'altezza dell'albero B e il costo computazionale.

Soluzione.

La prima idea che si può provare a perseguire è quella di inserire nell'albero con il maggior numero di nodi (senza perdere di generalità sia esso B_1) i nodi dell'altro albero uno ad uno. Sapendo che l'operazione di inserimento in un albero binario di ricerca ha un costo dell'ordine dell'altezza dell'albero si ha, nel caso peggiore, che il costo computazionale è:

$$O(h_1)+O(h_1+1)+O(h_1+2)+...+O(h_1+n_2-1) = n_2O(h_1)+O(1+2+...+n_2-1).$$

Poichè nel caso peggiore $O(h_1) = O(n_1)$ ed $O(1 + 2 + ... + n_2) = O(n_2^2)$, si ottiene che il costo di questo approccio è $O(n_1n_2) + O(n_2^2) = O(n_1n_2)$, essendo per ipotesi $n_1 > n_2$.

Tuttavia, questa soluzione non utilizza l'ipotesi che tutti gli elementi di B_1 siano minori di quelli di B_2. Per tentare di sfruttare questa ipotesi (è buona norma tenere presente che, se un'ipotesi c'è, allora serve a qualcosa!) possiamo pensare di appendere l'intero albero B_1 come figlio sinistro del minimo di B_2. Questo si può sempre fare facilmente perchè il minimo di un albero binario di ricerca è, per definizione, il nodo più a sinistra che non possiede figlio sinistro, pertanto è sufficiente settare un puntatore sulla radice di B_2, scendere verso il figlio sinistro finchè esso esista e, giunti al nodo che non ha figlio sinistro (che è il minimo), agganciargli a sinistra la radice di B_1. Osserviamo che questo procedimento è corretto perchè un

albero binario di ricerca ***non*** è necessariamente bilanciato, e quindi poco importa che l'altezza dell'albero risultante possa diventare $O(h_1 + h_2)$. Il costo di questo approccio è dominato dal costo della ricerca del minimo, cioè da $O(h_1)$ ed è, pertanto, da preferirsi al primo metodo proposto.