Algoritmi e Strutture Dati **Anno Accademico 2021–2022**

Primo Appello
8 Giugno 2022

SECONDA PARTE

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 11 punti)

Si consideri il seguente problema:

INPUT: un array A di n numeri tutti distinti tra loro tranne per una coppia, con $A[i] \in \{1, 2, 3, \ldots, n-1\}$ per ogni $i = 0, 1, \ldots, n-1$, ordinato in senso crescente.
OUTPUT: l'unico intero $k \in \{1, 2, \ldots, n-1\}$ che compare due volte in A.

1. Mostrare un esempio di input e corrispondente output del problema con un array di 10 elementi;
2. Descrivere brevemente a parole un algoritmo di costo computazionale $O(\log n)$ che risolva il problema;
3. Scrivere lo pseudocodice dell'algoritmo descritto precedentemente a parole;
4. Se $A[i] \in \{1, 2, 3, \ldots, n-2\}$ per ogni $i = 0, 1, \ldots, n-1$ e ci fossero in A esattamente due numeri $k_1 \neq k_2$ che compaiono due volte, come potrebbe essere modificato l'algoritmo presentato precedentemente per trovare il più grande tra k_1 e k_2, sempre in tempo $O(\log n)$? Rispondere a parole, non è necessario lo pseudocodice.

Esercizio 2. (max 10 punti)

Si consideri il seguente problema: dato in input un albero binario T realizzato con nodi e puntatori (campi $left, right, val$ con significato standard) che memorizza valori interi nei nodi, modificare l'albero copiando la chiave memorizzata nella radice dell'albero T nelle foglie.

1. Mostrare un esempio di input e corrispondente output del problema su un albero di 10 nodi;
2. Descrivere brevemente a parole un algoritmo che, usando la ricorsione, risolva il problema. Presentare esplicitamente il caso base e il caso ricorsivo della ricorsione;
3. Scrivere lo pseudocodice dell'algoritmo descritto precedentemente a parole;
4. Mostrare sull'esempio fornito precedentemente l'ordine delle chiamate ricorsive e della loro terminazione;
5. Studiare il costo computazionale dell'algoritmo proposto.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)

Presentare il problema dei cammini minimi da sorgente singola con archi di peso anche negativi. Presentare l'algoritmo di Bellman-Ford per la soluzione al problema (lo pseudocodice non è strettamente necessario, solo lo pseudocodice non è sufficiente senza spiegazioni) e discuterne il costo computazionale.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Esempio:

INPUT: $< 1, 2, 3, 4, 5, 6, 7, 8, 8, 9 >$ OUTPUT: 8

Dato che l'array è ordinato, per ottenere costo logaritmico, usiamo una variante della binary search. Per capire come decidere su quale metà invocare la ricorsione osserviamo la relazione tra i valori nell'array e i rispettivi indici. Guardando l'esempio

Indice	0	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	8	9

concludiamo che, in generale:

- le due occorrenze del numero ripetuto sono consecutive perché l'array è ordinato. Le due occorrenze si trovano quindi nelle posizioni r e $r+1$ tali che $0 \leq r \leq n-2$ e $1 \leq r+1 \leq n-1$;
- fino all'indice r abbiamo che $A[i] = i+1$ per ogni $i = 0, \ldots, r$;
- a partire dall'indice $r+1$ abbiamo che $A[i] = i$ per ogni $i = r+1, \ldots n-1$.

VERSIONE 1

Cerchiamo il valore in posizione r testando le seguenti due condizioni per l'indice centrale k di $A[i..j]$: $A[k] = k+1$ e $A[k+1] = k+1$.

Osserviamo che, poiché l'indice r non è mai l'ultimo dell'array, l'indice $k+1$ sarà anch'esso nel range degli indici di A. Inoltre, poiché il numero ripetuto esiste sicuramente (da specifiche dell'input) non capiterà di chiamare la ricorsione su una porzione di array nulla. Quindi:

- calcoliamo l'indice centrale k di $A[i..j]$ come nella ricerca binaria;
- se $A[k] = k+1$ e $A[k+1] = k+1$ allora abbiamo trovato r (che è $k+1$) e restituiamo $A[k]$ (questo è il caso base);
- altrimenti:
  - se $A[k] = k$ allora dobbiamo continuare a cercare a sinistra;
  - se $A[k] \neq k$, sarà necessariamente $A[k] = k+1$, e dobbiamo continuare a cercare a destra.

Osservazione: il motivo per cui in molti esercizi vi chiedo di descrivere l'algoritmo a parole è che spesso la trascrizione in pseudo-codice è immediata, come in questo caso.

```
CercaDoppione(A,i,j)
  k := ⌊ (i+j)/2 ⌋ // funziona anche se i = j, avremo k = i = j e il test del caso base sara‘ true
  if A[k] = k+1 AND A[k+1] = k+1 // caso base
     then return k+1
  if A[k] = k // caso ricorsivo
     then return CercaDoppione(A,i,k-1) // continuare a cercare a sinistra
     else return CercaDoppione(A,k+1,j) // continuare a cercare a destre
```

VERSIONE 2

Questa versione ha un codice che assomiglia molto alla prima versione (anche perché trattandosi di poche righe di codice la differenza tra le due versioni non può essere enorme), ma la spiegazione del perchè funziona è molto più complessa perché la terminazione può avvenire per motivi diversi e tutti devono essere analizzati per controllare che in ogni caso la terminazione sia corretta e non dimenticare nessun caso.

Per risolvere il problema possiamo andare a cercare l'ultimo indice r per cui vale $A[i] = i + 1$ oppure il primo per cui vale $A[i] = i$, che è $r + 1$. Data una porzione generica di A che inizia all'indice i e termina all'indice j tali che $i < j$ (cioè ci sono almeno due elementi):

- calcoliamo l'indice centrale k di $A[i..j]$ come nella ricerca binaria;
- se $A[k] = k$ allora dobbiamo continuare a cercare a sinistra (attenzione, se k è il primo per cui vale questa condizione - ovvero è l'indice per cui $k = r + 1$ - prendendo la porzione di sinistra "tagliamo fuori" questo elemento);
- se $A[k] \neq k$, sarà necessariamente $A[k] = k + 1$, e dobbiamo continuare a cercare a destra (attenzione, se k è l'ultimo per cui vale questa condizione - ovvero è l; indice per cui $k = r$ - prendendo la porzione di destra "tagliamo fuori" questo elemento).

Ci fermiamo quando $i \geq 1j$, ovvero quando abbiamo un solo elemento che sarà quello all'indice r o $r + 1$ a seconda dell'input, oppure quando gli indici si scambiano ($i > j$).

```
CercaDoppione(A,i,j)
  if i ≥ j // caso base
     then return A[i]
  k := ⌊ (i+j)/2 ⌋
  if A[k] = k
     then return CercaDoppione(A,i,k-1) // continuare a cercare a sinistra
     else return CercaDoppione(A,k+1,j) // continuare a cercare a destre
```

Osserviamo che in questa versione il caso base è legato solo alla dimensione della porzione di array e non viene fatto un check del contenuto di celle dell'array. Potrebbe succedere che k cada su uno dei numeri cercati, ma l'algoritmo non si ferma in questo caso.

La spiegazione del perché questa versione funziona mi è venuta più lunga del previsto, ma è un buon esercizio cercare di seguirla.

Osserviamo che le due occorrenze del numero doppio possono

1. rimanere insieme nella stessa porzione dell'array fino a che la porzione ha dimensione esattamente due elementi (e contiene i due doppioni);
2. oppure i due doppioni possono essere "separati", ovvero uno rimane nella porzione da analizzare e l'altro no. L'importante è che almeno uno dei due rimanga sempre nella porzione da analizzare fino alla penultima chiamata ricorsiva.

   Osserviamo che l'unico motivo per cui i due doppioni possono essere separati (ma che uno rimanga nella parte da analizzare mentre l'altro finisce nella parte scaratata) è che k cada su una delle due occorrenze.

   (a) Se $k = r$ (sull'occorrenza di sinistra), allora solo l'occorrenza di destra rimarrà nella porzione da analizzare.
   (b) Se $k = r + 1$ (sull'occorrenza di destra), allora solo l'occorrenza di sinistra rimarrà nella parte da analizzare.

Inoltre, osserviamo che non è possibile che entrambe le occorrenze vengano scartate contemporaneamente.

Per capire perché funziona, guardiamo qualche esempio di esecuzione con un array di 21 elementi (un pò più lungo del precedente per capire meglio).

- Esempio del caso 1. in cui le due occorrenze del doppione rimangono nella parte di array:
  INPUT

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	1	2	3	4	5	6	7	8	9	11	10	12	13	14	15	16	**17**	**17**	18	19	20

  Esecuzione:

  Primo $k = 10$ - andiamo a destra, nuovi indici $i = 11, j = 20$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20
A												12	13	14	15	16	**17**	**17**	18	19	20

  Secondo $k = 15$ - andiamo a destra, nuovi indici $i = 16, j = 20$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19	20
A																	**17**	**17**	18	19	20

  Terzo $k = 18$ - andiamo a sinistra, nuovi indici $i = 16, j = 17$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	**18**	19	20
A																	**17**	**17**			

  Quarto $k = 16$, andiamo a destra, nuovi indici $i = 17, j = 17$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**16**	17	18	19	20
A																		**17**			

  Abbiamo $i = j$ e restituiamo $A[i] = A[17] = 17$.

  Spiegazione generale: Se, a forza di ridurre la porzione di array su cui lavorare, finiamo per rimanere con una porzione lunga due elementi che contiene entrambe le ripetizioni, il prossimo k sarà l'indice più a sinistra che porterà ad andare a destra e a trovarsi sulla seconda occorrenza come unico elemento nella porzione. Verrà correttamente restituita la seconda occorrenza del doppione.

- Se le due occorrenze del doppione vengono "separate" e ne rimane solo una nella parte di array da analizzare, può succedere che rimanga la seconda (caso 2.a), quella di destra. In questo esempio succede alla terza chiamata ricorsiva:

  INPUT:

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	1	2	3	4	5	6	7	**8**	**8**	9	10	12	13	14	15	16	17	18	19	20	21

  Esecuzione:

  Primo $k = 10$ - andiamo a sinistra, nuovi indici $i = 0, j = 9$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20
A	1	2	3	4	5	6	7	**8**	**8**	9											

  Secondo $k = 4$ - andiamo a destra, nuovi indici $i = 5, j = 9$ - porzione rimanente:

Indice	0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A						6	7	**8**	**8**	9											

  Terzo $k = 7$ - andiamo a destra, nuovi indici $i = 8, j = 9$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	16	17	18	19	20
A									**8**	9											

  Abbiamo eliminato la prima occorrenza del numero doppio, quella a sinistra, perché k cadeva proprio al suo indice. Rimane "da sola" l'occorrenza di destra.

  Quarto $k = 8$ - andiamo a sinistra, nuovi indici $i = 8, j = 7$ - porzione rimanente:

Indice	0	1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	15	16	17	18	19	20
A																					

  Abbiamo $i > j$ e restituiamo $A[i] = A[8] = 8$

  Spiegazione generale: se eliminiamo la prima occorrenza del doppione perché k ci "cade sopra" ad un certo punto, da quel punto in poi rimarrà una porzione di array in cui tutti gli elementi sono uguali al loro indice e avremo sempre $i = r + 1$. Quindi tutte le ricorsioni successive sceglieranno la parte sinistra della porzione lasciando la seconda occorrenza del doppione come estremo sinistro (ovvero $i = r + 1$), fino a quando:

  - k cardà sulla seconda occorrenza del doppione, ovvero $i = k$, come in questo esempio. Alla ricorsione successiva avremo i invariato rispetto a prima, ma $j = k - 1 = i - 1$. Quindi avremo $i > j$ e i ancora sulla occorrenza di destra del doppione. Verrà correttamente restituita la seconda occorrenza del doppione in $A[i]$.
  - k cardà sulla posizione successiva alla seconda occorrenza del doppione, ovvero $k = i+1 = r+2$. Andando a sinistra ancora, si avrà una porzione di un elemento (il doppione) e verrà corettamente restituita la seconda occorrenza del doppione in $A[i]$ (abbiamo $i = j$).

- Se le due occorrenze del doppione vengono "separate" e ne rimane solo una nella parte di array da analizzare, può succedere che rimanga la prima (caso 2.b).

  INPUT

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	1	2	3	**4**	**4**	5	6	7	8	9	10	12	13	14	15	16	17	18	19	20	21

  Esecuzione: le prime chiamate ricosive si hanno con $k = 10, 4, 1$ (con $k = 4$ si elimina l'occorrenza di destra del doppione) e arriviamo alla situazione seguente: $i = 2, j = 3$.

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A			3	**4**																	

  Il prossimo k è 2 e otteniamo

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A				**4**																	

  Abbiamo $i = j$ e restituiamo correttamente $A[i] = A[3] = 4$.

  Spiegazione generale: è duale rispetto al quella del caso 2.a in cui ci troviamo alla fine con un elemento solo (ovvero quello che non capita nell'esempio del caso 2.a).

Per concludere, osserviamo che quando uno dei due doppioni viene eliminato, l'altro viene individuato correttamente. Poiché, come abbiamo osservato all'inizio, non è possibile che i due doppioni vengano scartati contemporaneamente, non accardà mai che entrambi vengano scartati e non trovati.

Punto 4. Nel caso in cui ci fossero due doppioni, il più grande si trova più a destra e, a partire dalla sua seconda occorrenza avremo che $A[i] = i - 1$:

Indice	0	1	2	3	4	5	6	7	8	9
A	1	2	2	3	4	5	6	6	7	8

Per risolvere il problema si lavora in modo analogo a quanto fatto prima, modificando opportunamente i test in modo da trovare l'ultimo valore per cui vale $A[i] = i$ o il primo per cui $A[i] = i - 1$.

VERSIONE 1 Modificare il test del caso base: $A[k] = k$ e $A[k + 1] = k$; modificare il test del caso ricorsivo: $A[k] = k - 1$.
VERSIONE 2 Modificare il test del caso ricorsivo: $A[k] = k - 1$

ERRORI PIÙ COMUNI:

- Sbagliare il codice della binary search chiamando la ricorsione su entrambe le metà dell'array e non solo su una parte. In questo caso il costo computazionale non risulta $O(\log n)$ ma lineare in n.
- Sbagliare il controllo per decidere su quale metà invocare la chiamata ricorsiva.
- Utilizzare sempre 1 o n per calcolare l'indice centrale della porzione di array da analizzare

- Non indicare la chiamata principale
- Dimenticare i `return` davanti alle chiamate ricorsive (il risultato non verrà mai restituito)

SOLUZIONE Esercizio 2.

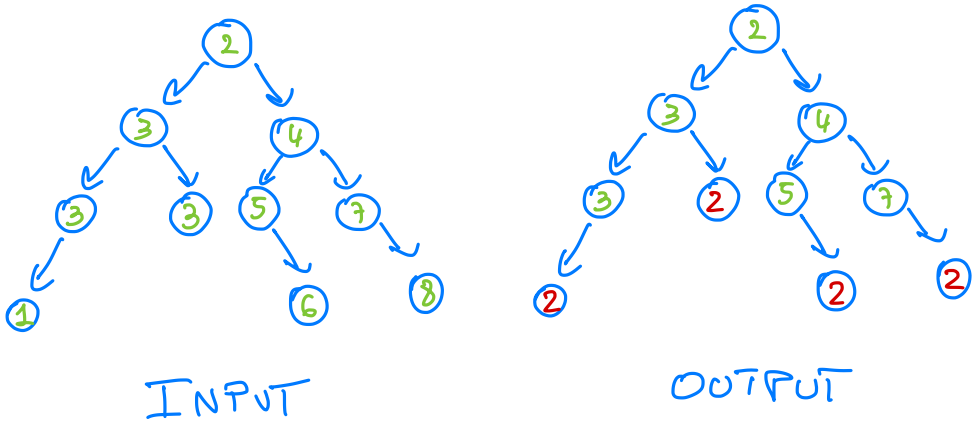

Figura 1: Esempio **punto 1** esercizio 2

Per risolvere l'esercizio si utilizza una procedura DFS che va a modificare l'albero. La procedure prende in input il puntatore ad un albero t e un valore k che deve essere copiato nelle foglie. Il valore di k viene definito nella procedura principale.

La procedura principale controlla che T non sia vuoto o una foglia (in questi casi non si deve fare niente), altrimenti invoca la chiamata ricorsiva impostando il valore di k corretto:

```
CambiaFoglie(T)
  if T ≠ NIL AND T.left ≠ NIL AND T.right ≠ NIL
     then Copia(T,T.val)
```

Ci possono essere almeno due versioni diverse della procedura `Copia` a seconda dei casi base che si considerano.

Versione 1

- **Caso Base:** l'albero t è una foglia: si copia il valore k dato in input nel campo val della foglia.
- **Caso Ricorsivo:** l'albero t non è una foglia: si richiama ricorsivamente la procedura sui figli non vuoti utilizzando lo stesso valore k ricevuto in input.

```
Copia(t,k)
  if t.left = NIL AND t.right = NIL // caso base
     then
       t.val := k
     else // caso ricorsivo
       if t.left ≠ NIL then Copia(t.left,k)
       if t.right ≠ NIL then Copia(t.right,k)
```

Versione 2

- **Casi Base:** (1) l'albero t è vuoto: non si fa niente. (2) L'albero t è una foglia: si copia il valore k dato in input nel campo val della foglia.
- **Caso Ricorsivo:** l'albero t non è una foglia: si richiama ricorsivamente la procedura sui figli utilizzando lo stesso valore k ricevuto in input

```
Copia(t,k)
  if t ≠ NIL // caso base (1)
     then
       if t.left = NIL AND t.right = NIL // caso base (2)
          then
            t.val := k
          else // caso ricorsivo
            Copia(t.left,k)
            Copia(t.right,k)
```

OSSERVAZIONE: in questa versione la procedura principale `CambiaFoglie(T)` può essere sostituita da `Copia(T,T.val)`.

Punto 5. Il costo computazionale è lo stesso di una visita DFS, perché si visitano tutti i nodi una volta sola e per ogni nodo il costo del lavoro è costante. Quindi il costo computazionale è $O(n)$, dove n è il numero di nodi dell'albero.

ERRORI PIÙ COMUNI:

- Dimenticare qualche foglia nell'esempio al punto 1
- **Punto 4:** Non indicare le chiamate sugli alberi vuoti (figli delle foglie o destro/sinistro di un nodo interno) nei casi in cui il codice dato come soluzione effettua anche quelle chiamate. Attenzione: alcune versioni del codice non mettevano in else le chiamate ricorsive rispetto al test foglia, quindi in questo caso le chiamate sui figli vuoti delle foglie vengono invocate. Esempio:

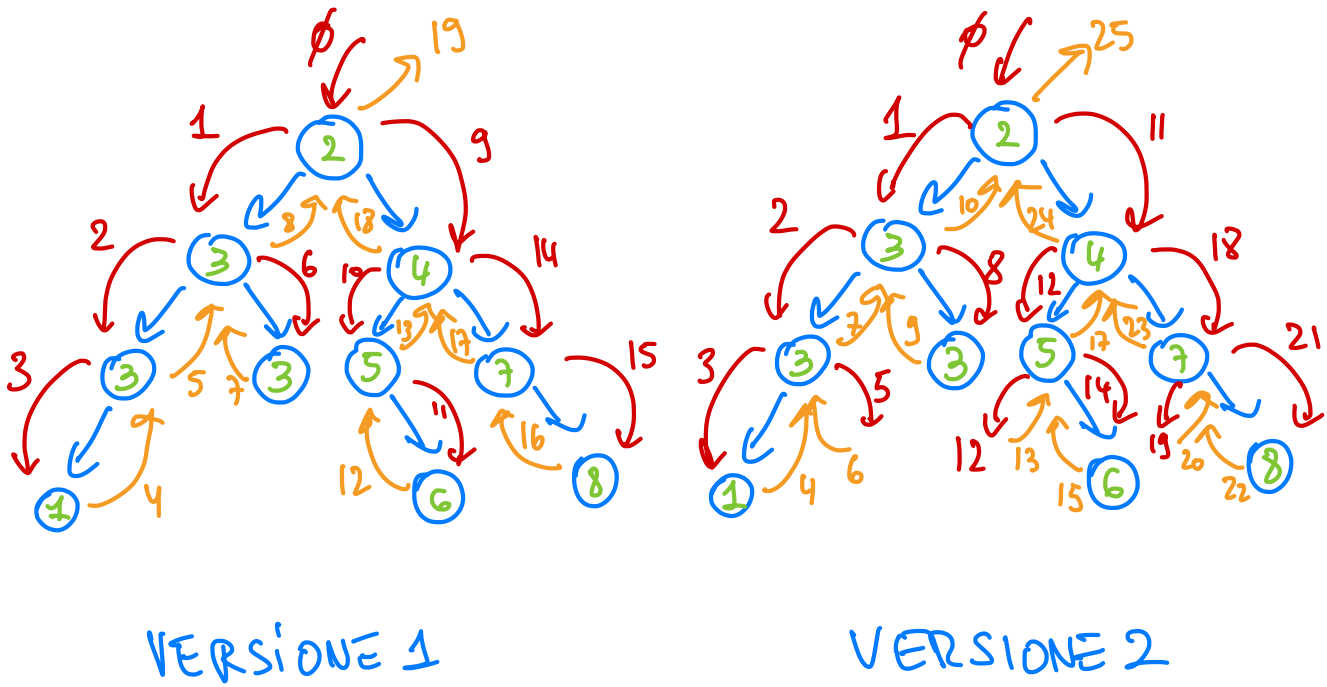

Figura 2: **Punto 4** esercizio 2

```
Copia(t,k)
  if t ≠ NIL // caso base (1)
     then
       if t.left = NIL AND t.right = NIL // caso base (2)
          then t.val := k
       // caso ricorsivo - ricorsione invocata anche se ci troviamo in una foglia
       Copia(t.left,k)
       Copia(t.right,k)
```

- Non indicare la chiamata principale quando questa serve per instanziare il parametro relativo al valore da copiare nelle foglie
- Non esplicitare il caso base e il caso ricorsivo come richiesto al punto 2; dimenticare un caso base (foglia o albero vuoto), a seconda delle scelte fatte nel codice; non dire cosa fare nei due casi; indicare le azioni da fare nella foglia (caso base) nel caso ricorsivo
- Sbagliare il codice della DFS.
- Indicare costo computazionale quadratico.
- Copiare nelle foglie il valore del padre e non quello nella radice dell'albero passato in input al problema. È un altro esercizio.

Secondo Appello
22 Giugno 2022

SECONDA PARTE (la traccia non deve essere consegnata)

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 10 punti)

Si consideri il seguente problema:

INPUT: Un albero binario realizzato con nodi e puntatori (campi $left, right, key$ con significato standard) che memorizza valori interi nei nodi
OUTPUT: Il numero di foglie la cui chiave è il doppio di quella del padre

1. Mostrare un esempio di input e output del problema con un albero con almeno 8 foglie.
2. Fornire una spiegazione a parole di un algoritmo ricorsivo che risolva il problema. Indicare esplicitamente il caso base e il caso ricorsivo e quali azioni intraprendere in entrambi i casi.
3. Fornire lo pseudocodice dell'algoritmo descritto a parole precedentemente.
4. Studiare il costo computazionale dell'algoritmo proposto.
5. Rispondere a parole: supponiamo che i nodi dell'albero abbiano identificativi $\{1, 2, \ldots, n\}$ e l'albero sia rappresentato con un vettore dei padri P tale che $P[i]$ contiene una coppia (p, key): p è l'identificativo del padre del nodo i e key è il valore memorizzato nel nodo i. Come si potrebbe procedere per risolvere lo stesso problema? La soluzione non deve essere necessariamente ricorsiva.

Esercizio 2. (max 11 punti) Sia $G = (V, E)$ un grafo non diretto e connesso. Dati due vertici $u, v \in V$, la **distanza** tra u e v è il numero di archi del cammino più breve che va da u a v. Si definisca **diametro** del grafo G la massima distanza tra due vertici di V. Rispondere ai seguenti quesiti:

1. Fornire un esempio di grafo con almeno 5 nodi e almeno 8 archi. Indicare il valore del diametro del grafo dato in esempio e mostrare quale cammino determina tale valore.
2. Spiegare perché la DFS non è lo strumento adatto per calcolare il diametro di G.
3. Descrivere brevemente a parole e fornire lo pseudocodice di un algoritmo per calcolare **il valore** del diametro di un grafo $G = (V, E)$ connesso e non diretto, tale che $V = \{1, 2, \ldots, n\}$.
4. Studiare il costo computazionale dell'algoritmo proposto.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)
Dare la definizione di albero binario completo e perfettamente bilanciato. In generale, (1) quante sono le foglie di un albero binario completo e perfettamente bilanciato di altezza h? (2) E quanti nodi? Dimostrare una delle due affermazioni.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

L'algoritmo ricorsivo utilizza una strategia DFS simile a quella utilizzata per contare il numero di foglie di un albero. La funzione principale `Conta(T)` invoca la DFS `ContaFoglie` solo se l'albero dato in input non è vuoto e non è una foglia (in questi casi la risposta è zero).

`ContaFoglie` prende in input il puntatore t alla radice di un albero e un valore k che è la chiave del padre di t e restituisce il numero di foglie che soddisfano la proprietà nel sottoalbero radicato in t.

- **Caso base:** l'albero t è una foglia. Confrontiamo il valore k dato in input con il valore $t.key$. Se $t.key = 2k$ allora restituiamo 1, altrimenti restituiamo zero
- **Caso ricorsivo:** l'albero t non è una foglia. Chiamiamo ricorsivamente la funzione sui figli di t non vuoti, con parametro $k = t.key$. Restituiamo la somma dei valori restituiti dalle due chiamate.

```
Conta(T)
  if T ≠ NIL AND T.left ≠ NIL AND T.left ≠ NIL
     then return ContaFoglie(T,T.key)
     else return 0
```

```
ContaFoglie(t,k)
   if t.left = NIL AND t.right = NIL
      then // caso base
        if t.key = 2k
           then return 1
           else return 0
      else // caso ricorsivo
        s := 0
        d := 0
        if t.left ≠ NIL then s := ContaFoglie(t.left, t.key)
        if t.right ≠ NIL then d := ContaFoglie(t.right, t.key)
        return s + d
```

Osservazione: la chiamata principale deve essere fatta con $T.key$, per includere anche il caso in cui la radice abbia solo due foglie come figli.

Si può scegliere di avere due casi basi, includendo anche il caso base albero vuoto: in questo caso si deve far restituire zero (ed effettivamente, nell'albero vuoto ci sono zero foglie che soddisfano la proprietà). In questa variante, non è più necessario il controllo sul puntatore vuoto prima di invocare la funzione ricorsivamente sui figli.

```
ContaFoglie(t,k)
   if t = NIL then return 0 // caso base 1
   if t.left = NIL AND t.right = NIL
      then // caso base 2
        if t.key = 2k
```

```
        then return 1
        else return 0
    else // caso ricorsivo
      return ContaFoglie(t.left, t.key) + ContaFoglie(t.right, t.key)
```

Chiamata principale `Contafoglie(T,T.key)`.

Questa seconda versione fa meno controlli all'interno della funzione, ma fa più chiamate ricorsive (anche se rimangono sempre lineari nel numero di nodi).

Variante: Contafoglie può prendere in input il puntatore al padre invece che il valore nel padre.

Versione 2.

A differenza della versione precedente, in questo caso non scendiamo fino alle foglie, ma ci fermiamo ai padri di foglie: controlliamo se i figli di un nodo sono foglie o meno, nel caso in cui lo siano testiamo la condizione e teniamo traccia dei test che danno esito positivo. La ricorsione viene invocata solo su figli che non sono foglie.

Rispetto alla precedente, in questa versione sono richiesti molti più confronti per controllare i puntatori vuoti ed evitare di andare a leggere il campo di un nodo che non esiste.

Utilizziamo una funzione principale in cui controlliamo se T è vuoto, oppure una foglia. In questi casi restituiamo zero, altrimenti viene invocata la funzione ricorsiva che lavora nel modo seguente:

- **Caso base:** la radice di t ha due foglie come figli. Per ognuna controlliamo se la chiave della foglia è il doppio di $t.key$ e restituiamo 0, 1 o 2 in base a quanti controlli danno esito positivo.
- **Caso ricorsivo:** la radice di t è un nodo interno con almeno un figlio che non è una foglia. Se i due figli sono nodi interni, allora richiamiamo ricorsivamente la funzione sui figli e restituiamo la somma dei risultati delle chiamate. Se uno dei due figli è una foglia, controlliamo se la chiave di quest'ultima è il doppio di $t.key$. In caso affermativo restituiamo 1 più il risultato della chiamata sull'altro figlio, altrimenti restituiamo solo il risultato della chiamata sull'altro figlio.

Per comodità scriviamo una funzione per controllare se un nodo è una foglia o meno. Trattiamo anche il caso in cui l'albero sia vuoto e facciamo restituire falso.

Utilizziamo anche una seconda funzione che, presi due puntatori a due nodi (padre e figlio nell'albero) controlla la se chiave del figlio è il doppio della chiave del padre.

```
Conta(T)
  if T ≠ NIL AND T.left ≠ NIL AND T.left ≠ NIL
     then return ContaFoglie(T)
     else return 0


IsLeaf(t)
  if t = NIL then return false
  if t.left = NIL AND t.right = NIL
     then return true
     else return false
```

```
Check(t1,t2) // t1 e' il padre, t2 e' il figlio
  if 2 * t1.key = t2.key
     then return 1
     else return 0


ContaFoglie(t)
   if IsLeaf(t.left) AND IsLeaf(t.right)
      then // caso base
        return Check(t,t.left) + Check(t,t.right)
      else // caso ricorsivo
        s := 0
        d := 0
        if t.left ≠ NIL
        then // c'e' un figlio sinistro
          if IsLeaf(t.left)
             then s:= Check(t,t.left) // il figlio sinistro e' una foglia
             else s := ContaFoglie(t.left) // a sinistra c'e' un sottoalbero con piu' di un nodo
        if t.right ≠ NIL
        then // c'e' un figlio destro
          if IsLeaf(t.right)
             then d:= Check(t,t.right) // il figlio destro e' una foglia
             else d := ContaFoglie(t.right) // a destra c'e' un sottoalbero con piu' di un nodo
        return s + d
```

Osservazione: il codice si può alleggerire "mescolando" il caso base con il caso ricorsivo, ricordandosi gli esiti dei controlli sui figli nel caso base, nel caso in cui ci sia una foglia. Si può anche utilizzare un contatore solo, invece che uno per il sottoalbero destro e uno per il sottoalbero sinistro.

Versione 3

Utilizziamo un contatore globale inizializzato nella funzione principale e incrementato da una procedura ricorsiva `ContaFoglie` quando necessario (attenzione, la procedura non restituisce nessun valore esplicitamente). Bisogna fare molta attenzione a gestire il contatore globale correttamente.

`ContaFoglie` prende in input il puntatore t alla radice di un albero e un valore k che è la chiave del padre di t:

- **Caso base:** l'albero t è una foglia. Confrontiamo il valore k dato in input con il valore $t.key$. Se $t.key = 2k$ allora incrementiamo di uno il contatore globale, altrimenti non facciamo niente.
- **Caso ricorsivo:** l'albero t non è una foglia. Chiamiamo ricorsivamente la funzione sui figli di t non vuoti, con parametro $k = t.key$.

```
Conta(T)
  count := 0
  if T ≠ NIL AND T.left ≠ NIL AND T.left ≠ NIL
     then
       ContaFoglie(T,T.key)
  return count
```

```
ContaFoglie(t,k)
   if t.left = NIL AND t.right = NIL
      then // caso base
        if t.key = 2k
           then count := count + 1
      else // caso ricorsivo
        if t.left ≠ NIL then ContaFoglie(t.left, t.key)
        if t.right ≠ NIL then ContaFoglie(t.right, t.key)
```

ERRORI PIÙ COMUNI:

- Sbagliare esercizio: contare il numero di nodi (anche interni) che godono della proprietà
- Non indicare esplicitamente il caso base e il caso ricorsivo come indicato nella traccia, non dire quali sono le operazioni da fare nei due casi.
- Dimenticare un caso base o parlare del caso base foglia nel caso ricorsivo
- Scrivere una funzione ricorsiva ma invece di restituire la somma delle chiamate sui sottoalberi, invocare la funzione sul sottoalbero destro e sinistro, senza assegnare il risultato delle chiamate ricorsive a nessuna variabile locale e senza sommarne i risultati (tutto il lavoro fatto nelle chiamate ricorsive viene perso!)
- Utilizzare un contatore globale in modo errato. Se si usa un contatore globale ci deve essere una funzione principale in cui questo contatore viene inizializzato e che ne restituisce il valore alla fine. Il contatore viene poi aggiornato da una procedura ricorsiva. NON è possibile inizializzare a zero un contatore in un'unica funzione ricorsiva e aggiornare questo contatore nella funzione come se fosse una variabile globale. Il risultato sarà sempre solo il valore del contatore dell'ultima chiamata.
- Nel discutere il costo computazionale: non basta dire che il costo è $O(n)$ perché si visitano tutti i nodi una volta sola. È necessario anche indicare quanto si paga per la visita di ogni nodo. Supponiamo, infatti, che il lavoro svolto in ogni nodo ci costa $O(n)$, allora il costo totale sarebbe $O(n) \cdot O(n) \in O(n^2)$ e non $O(n)$. Nel caso di questo esercizio, il costo del lavoro in ogni nodo è costante (cioè $O(1)$, ovvero non dipende dal numero di nodi) e quindi il costo totale è effettivamente $O(n)$.

SOLUZIONE Esercizio 2.

ATTENZIONE: Non ho tempo di scrivere le soluzioni più dettagliate adesso, ma inizio a pubblicare una versione PRELIMINARE che integrerò (spero) prima della visione dei compiti)

Il problema chiedeva di trovare la distanza massima tra due nodi del grafo, dove la distanza è il cammino minimo tra due nodi. Quindi è necessario trovare il massimo tra i cammini minimi tra i nodi del grafo. Per farlo si possono utilizzare gli algoritmi visti a lezione per i cammini minimi, nel modo opportuno.

Per risolvere il problema la DFS non è uno strumento utile perché non permette di calcolare i cammini più brevi tra nodi, anzi, generalmente trova cammini molto più lunghi dei minimi perché lavora allontanandosi sempre di più dalla sorgente.

Versione 1

L'algoritmo di Floyd-Warshall trova i cammini minimi tra ogni coppia di nodi del grafo. Il diametro è il valore massimo presente nella matrice restituita dall'algoritmo. Osserviamo che, poiché il grafo è connesso e indiretto, nella matrice non saranno presenti valori a $+\infty$ (perché esiste un cammino, e quindi anche quello minimo, tra ogni coppia di nodi).

Il costo computazionale è quello dell'algoritmo di Floyd-Warshall, infatti la ricerca del massimo nella matrice ha costo inferiore a non incide su quello totale.

Versione 2

La BFS può essere utilizzata per calcolare la distanza dai nodi del grafo da una sorgente. Invocarla una volta sola non è sufficiente per risolvere il problema, ma la si deve invocare a partire da ogni nodo del grafo, per ogni nodo calcolare la distanza massima e, infine, calcolare il massimo tra le distanza massime.

Il costo computazionale è dato da n volte il costo della BFS. Anche in questo caso il calcolo dei massimi non aumenta il costo computazionale rispetto a quello della BFS.

Osservazione: si può evitare la chiamata sull'ultimo nodo, le sue distanze minime sono già state calcolate dagli altri nodi.

Osservazione: Anche l'algoritmo di Dijkstra calcola i cammini minimi in un grafo con archi pesati, ma in questo caso basta la BFS. Inoltre se si usa Dijkstra (o Bellman-Ford, che però in questo caso serve ancora meno perché i pesi sono positivi) è necessario specificare i costi degli archi esplicitamente.

ERRORI PIÙ COMUNI:

- Sbagliare l'esempio indicando come diametro un cammino non minimo e, la cosa strana, cercare di calcolare il diametro usando un algoritmo per cammini minimi.
- Non capire quale valore fosse richiesto calcolare e/o cercare di calcolare i cammini massimi (la distanza è un cammino minimo)
- Dimenticare la funzione costo quando si chiamano algoritmi visti a lezione (Dijkstra o Floyd-Warshall)
- Invocare la BFS o Dijkstra una volta sola da un nodo solo, iniziate a cercare un esempio in cui non funziona
- Sbagliare a riportare i costi computazionali degli algoritmi visti a lezione (BFS, Dijkstra, Floyd-Warshall)

Algoritmi e Strutture Dati **Anno Accademico 2021–2022**

Terzo Appello
20 Luglio 2022

SECONDA PARTE (la traccia non deve essere consegnata)

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 10 punti)

Si consideri il seguente problema:

INPUT: Un array A ordinato che memorizza n valori interi (sia positivi che negativi) tutti distinti.
OUTPUT: Stampa di tutte le coppie di indici (i, j) tali che $0 \leq i, j \leq n-1$, con $i \neq j$, per cui vale $A[i] = 3 \cdot A[j]$ se esistono, la coppia $(-1, -1)$ altrimenti.

1. Mostrare un esempio di input e output del problema con array A contenente sia valori positivi che negativi.
2. Fornire una spiegazione a parole di un algoritmo che risolva il problema in tempo **inferiore** a quadratico in n.
3. Fornire lo pseudocodice dell'algoritmo descritto a parole precedentemente.
4. Studiare il costo computazionale dell'algoritmo proposto.

Esercizio 2. (max 8 + 3 punti)

INPUT: Un albero binario T realizzato con nodi e puntatori (campi $val, left$ e $right$ con significato standard), e un puntatore t ad un nodo dell'albero.
OUTPUT: Il livello a cui si trova il nodo t nell'albero, oppure -1 se il nodo non si trova nell'albero.

(8 punti):

1. Mostrare un esempio di input e output del problema.
2. Fornire una spiegazione a parole di un algoritmo ricorsivo che risolva il problema, indicando **esplicitamente** il caso base e il caso ricorsivo **E** quali azioni intraprendere in entrambi i casi.
3. Fornire lo pseudocodice dell'algoritmo descritto a parole precedentemente.
4. Studiare il costo computazionale dell'algoritmo proposto.

(+ 3 punti): sfruttare la soluzione fornita precedentemente per risolvere il seguente problema (fornire lo pseudocodice): dato un albero binario T realizzato con nodi e puntatori, e due puntatori t_1 e t_2 a due nodi dell'albero, restituire VERO se t_1 e t_2 sono cugini, FALSO altrimenti. Due nodi sono cugini se sono allo stesso livello nell'albero ma hanno padri diversi.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)
Spiegare come funziona la BFS su grafi, introducendo il problema approcciato dalla BFS, scrivendo e commentando lo pseudocodice. Come può essere utilizzata la BFS per calcolare l'albero dei cammini minimi da sorgente singola nel caso in cui tutti i pesi degli archi siano uguali?

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Esempio: INPUT:

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	-10	-9	-7	-6	-3	-2	0	1	2	3	5	7	13	14	15	16	17	19	21	35	40

OUTPUT: (1,4), (3,5), (14,10), (11,18)

Osserviamo che:

- Se $A[i] > 0$, allora l'indice j per cui $A[j]$ è un terzo di $A[i]$ (ovvero l'indice da cercare, poiché $A[i]/3 = A[j]$) si trova necessariamente alla sinistra di i, perchè l'array è oridnato (quindi avremo $j < i$).
- Per simmetria, se $A[i] < 0$, allora j si trova alla destra di i (quindi avremo $j > i$).
- Se $A[i] = 0$, allora $j = i$.

Poiché gli indici sono tutti diversi, dato i, se j esiste, allora è unico.

Inoltre, se abbiamo $n \in \{0, 1\}$ allora non possono esistere coppie di indici i, j.

Versione 1.

Per ogni indice i dobbiamo cercare l'indice j per cui $A[j] = A[i]/3$. Poiché l'array è ordinato, possiamo utilizzare una binary serach per determinare se j esiste o meno in tempo logaritmico in n, dato i. Ricordiamo che la binary search restituisce l'indice a cui si trova la chiave cercata, se esiste, altrimenti restituisce -1. Utilizziamo un flag per ricordare se almeno una volta l'indice j sia stato trovato. In caso negativo, stampiamo $(-1, -1)$.

```
StampaIndici(A,n)
  if n < 2 then print (-1,-1)
  trovato := false
  for i = 0 to n-1 do
      j := BinarySearch(A,0,n-1,A[i]/3)
      if i ≠ j then
         print (i,j)
         trovato := true
  if NOT trovato then print (−1,−1)
```

Il costo di questo algoritmo nel caso peggiore è n volte quello della binary search, quindi $O(n \log n)$ che è inferiore a quadratico in n.

Osservazione. Il ciclo for deve comprendere sia il primo che l'ultimo elemento di A. Infatti, nel caso in cui gli elementi in A fossero tutti positivi, per $i = n-1$ ci potrebbe essere un $0 \leq j < n-1$ per cui $A[n-1] = 3A[j]$ e, analogamente, nel caso in cui tutti gli elementi in A fossero negativi, per $i = 0$ ci potrebbe essere un $0 < j \leq n-1$ tale che $A[0] = 3A[j]$.

Variante 1. Sfruttando l'osservazione fatta all'inizio, possiamo chiamare la binary search solo su una parte dell'array e non su tutto. Dal punto di vista asintotico il costo computazionale asintotico non cambia, ma in pratica si riduce il numero di chiamate ricorsive di binary search.

Possiamo anche controllare l'estremo corretto dell'array per evitare di invocare la binary search inutilmente. Per esempio: se $A[i] > 0$, allora j va cercato a sinistra, ma se $A[i]/3 < A[0]$, allora sicuramente j non esiste. Analogamente per il caso $A[i] > 0$. Inoltre, evitando di includere l'indice i nella porzione coinvolta dall'invocazione di binary search, non potrà mai succedere che i sia uguale a j e quindi non è necessario fare nessun controllo per evitare di stampare la coppia (i, i).

```
StampaIndici(A,n)
  if n < 2 then print (-1,-1)
  trovato := false
  for i = 0 to n-1 do
      if A[i] > 0 AND A[i]/3 ≥ A[0] then
         j := BinarySearch(A,i-1,0,A[i]/3)
         if j > -1 then print (i,j)
         trovato := true
      if A[i] < 0 AND A[i]/3 ≤ A[n-1] then
         j := BinarySearch(A,i+1,n-1,A[i]/3)
         if j > -1 then print (i,j)
         trovato := true
  if NOT trovato then print (-1,-1)
```

Versione 2.

Consideriamo per un attimo la porzione di array che contiene valori negativi: si osserva che, dopo aver trovato una coppia (i, j), se si scorre l'array da sinistra verso destra, la prossima coppia (i', j') avrà necessariamente $i' > i$ e $j' > j$. Per esempio: (1,4), (3,5). Analogamente, se si scorre l'array da destra verso sinistra, e ci si concentra solo sulla parte che contiene numeri positivi, una volta trovata una coppia (i, j), la successiva coppia (i', j') avrà necessariamente $i' < i$ e $j' < j$. Per esempio: (11,18), (14,10).

Quindi, possiamo scorrere l'array da sinistra a destra fino a quando $A[i] < 0$ facendo avanzare j senza farlo tornare indietro su posizioni già visitate (al più non lo facciamo avanzare). Successivamente possiamo scorrere l'array da destra a sinistra fino a quando $A[i] > 0$ facendo andare indietro j senza farlo tornare avanti su posizioni già visitate ((al più non lo facciamo indietreggiare). In questo modo si evita anche di visitare un'eventuale cella contenente zero.

Consideriamo di nuovo la porzione di array che contiene valori negativi e stabiliamo il modo in cui fare avanzare gli indici i e j. Si possono verificare i casi seguenti:

- La coppia (i,j) viene stampata: allora sia i che j devono avanzare di uno (essendo tutti i valori distinti non potrà mai capitare che compaiano di nuovo in un'altra coppia).
- A[i]/3 > A[j]: vuol dire che il valore che cerchiamo potrebbe ancora essere ad un indice j più grande, quindi dobbiamo incrementare j lasciando i invariato. Nell'esempio: se i=1 e j= 3, abbiamo A[1]/3 = -9/3 = -3 > -6 = A[3]. Il valore -3 potrebbe esistere alla sinistra di -6.
- A[i]/3 < A[j]: vuol dire che abbiamo superato il valore che stiamo cercando e quindi non esiste. Dobbiamo fare avanzare i, mentre j deve rimanere invariato perché A[j] potrebbe essere uguale a A[i+1]/3. Nell'esempio: se

i=2 e j= 5, abbiamo A[2]/3 = -2,3 (periodico) $<$ -2 = A[5]. Il valore -2,3 periodico non può esistere alla sinistra di -2, quindi non è più possibile trovare un indice per i=2 a partire dal valore corrente di j=2.

Si ragiona in modo analogo (ma simmetrico) per la porzione di array che contiene valori positivi.

```
StampaIndici(A,n)
  if n < 2 then print (-1,-1)
  trovato := false
  // scansione da sinistra a destra sui valori negativi
  i := 0
  j := 1
  while A[i] < 0 AND j ≤ n − 1 AND A[j] < 0 do
    if A[j] = A[i]/3 then // trovato indice j
       print (i,j)
       trovato := true
    if A[j] ≥ A[i]/3 then i := i+1 // abbiamo trovato j o non esiste j per questo i, quindi incrementiamo i
    if A[j] ≤ A[i]/3 then j := j+1 // abbiamo trovato j o j potrebbe ancora esistere, incrementiamo solo j
  // scansione da destra a sinistra sui valori positivi
  i := n-1
  j := n-2
  while A[i] > 0 AND j ≥ 0 AND A[j] > 0 do
    if A[j] = A[i]/3 then
       print (i,j)
       trovato := true
    if A[j] ≤ A[i]/3 then i := i-1
    if A[j] ≥ A[i]/3 then j := j-1
  if NOT trovato then print (-1,-1)
```

Osservazione: Si può inserire un ulteriore controllo: se A[i] non è un multiplo di 3 (A[i] $\mod 3 \neq 0$) si può direttamente fare avanzare i alla posizione successiva.

Costo computazionale: l'indice i scorre tutto l'array (escludendo al più un solo indice k per cui A[k] =0). Fissato i, l'indice j scorre una porzione di array A[j1..j2] e per ogni indice tra j1 e j2 (estremi inclusi) viene eseguito un solo confronto con A[i]. Inoltre, la porzione A[j1..j2] ha al massimo i due estremi in comune con le porzioni analizzate per $i-1$ e $i+1$. Quindi, per ogni j, A[j] viene coinvolto in al più tre confronti (se j1=j2). Quindi, il costo complessivo è $O(n)$.

ESERCIZIO: eseguire l'algoritmo di questa soluzione sull'esempio dato facendo vedere come si spostano gli indici i e j.

ERRORI PIÙ COMUNI:

- Nell'esempio: indicare come output le coppie di valori agli indici i e j e non le coppie di indici
- Fornire una soluzione quadratica che, con due for annidati o un while annidato in un for esterno, testa tutte le coppie di elementi. Anche escludendo la porzione prima o dopo l'indice i, il numero delle coppie è comunque $\Omega(n)$ nel caso peggiore. ESERCIZIO: mostrare un esempio.
- Invocare la binary search ricorsiva vista a lezione sbagliando i parametri. Esempio: dimenticando uno degli estremi della porzione di array da analizzare, oppure il valore da cercare.

- Cercare j tale che $3A[i] = A[j]$ e stampare (i, j) invece di (j, i).
- Riscrivere la binary search in modo che, presa tra gli input la chiave $A[i]$, restituisca l'indice di $A[i]/3$ e sbagliare a scrivere il codice della binary search (o, in alcuni casi, cercare $3A[i]$).
- Usare la binary search sempre e solo alla destra o alla sinistra di i.

SOLUZIONE Esercizio 2.

Versione 1

Utilizziamo una funzione ricorsiva simil DFS che prende in input il puntatore ad un nodo dell'albero t e quello dato in input da cercare t', più un valore k che indica il livello di t nell'albero dato in input T e restituisce il livello di t' nell'albero T.

- **Casi base:** (1) il nodo t' è uguale al nodo t, in questo caso si restituisce il valore k; (2) l'albero t è vuoto, si restituisce -1 per indicare che t' non è stato trovato.
- **Caso ricorsivo:** t non è vuoto e non è uguale a t': si continua a cercare t' nei sottoalberi radicati in t ricorivamente sui figli non vuoti aumentando di uno il valore di k. Al ritorno della ricorsione, si restituisce il massimo dei due valori. Se t' non è stato trovato, entrambe le chiamate restituiranno -1 e quindi anche il massimo restituito sarà -1; altrimenti t' sarà solo in uno dei due sottoalberi, La chiamata in questo sottoalbero resituirà un valore maggiore o uguale a zero, mentre l'altro -1, quindi il valore restituito sarà sicuramente il livello di t'.

Osservazione. Per capire se i nodo corrente è quello che stiamo cercando confrontiamo i puntatori. Confrontare i valori memorizzati nel campo key non è corretto perché non è detto che le chiavi siano uniche.

```
livello(t,t',k)
  if t = NIL then return -1
  if t = t' then return k
  if t.left not = NIL then l := livello(t.left,t',k+1)
  if t.right not = NIL then r := livello(t.right,t',k+1)
  return max(l,r)
```

Chiamata principale: `livello(T,t,0)`.

ATTENZIONE: in questo caso la chiamata principale è NECESSARIA perché non c'è una funzione principale. Senza la chiamata principale non sarebbe noto quali sono i parametri con cui debba essere invocata per trovare la soluzione al problema. Se ci fosse una funzione principale che invoca questa funzione, allora l'invocazione indicherebbe necessariamente quali sono i parametri corretti per risolvere il problema.

Per decidere se i due nodi sono cugini, possiamo procedere nel modo seguente:

- Calcolo il livello di t_1 in T utilizzano la funzione progettata precedentemente
- Calcolo il livello di t_2 in T utilizzano la funzione progettata precedentemente
- Nel caso in cui i livelli siano lo stesso (e diverso da -1), controllo se i due nodi hanno lo stesso padre o meno e rispondo di conseguenza, altrimenti rispondo subito falso.

Utilizziamo una funzione principale che esegue le operazioni indicate e che chiama una nuova funzione che controlla se due nodi sono fratelli. La funzione è ricorsiva simil DFS, prende in input un puntatore t ad un nodo di T e i puntatori a due nodi t_1 e t_2 e restituisce vero se sono fratelli, falso altrimenti.

- **Casi base:** (1) il nodo t' è vuoto, si restituisce true (se t_1 e t_2 non sono stati trovati non sono fratelli). (2) t è padre di t_1 e t_2, si restituisce vero.
- **Caso ricorsivo:** t non è vuoto e non è padre di t_1 e t_2: si cercano t_1 e t_2 in entrambi i sottoalberi e si restituisce l'or delle chiamate ricorsive (se ci sono e sono fratelli saranno solo in uno dei due sottoalberi).

```
Check(T,t1,t2)
   if T≠ NIL then // e anche i puntatori
      l1 := livello(T,t1,0)
      l2 := livello(T,t2,0)
      if l1 > -1 AND l2 > -1 AND l1 = l2 then return not samefather(T,t1,t2)
   return False
```

```
samefather(t, t1, t2)
  if t = nil then return false
  if (t.left = t1 AND t.right = t2) OR (t.left = t2 AND t.right = t3) then return true
  return samefather(t.left, t1,t2) OR samefather(t.right,t1,t2)
```

Costo computazionale: il costo computazionale è quello di una DFS che visita, nel caso peggiore in cui il nodo non c'è o è l'ultimo ad essere visitato, tutti i nodi dell'albero esattamente una volta sola ed esegue un lavoro costante per ogni nodo (escluse le chiamate ricorsive). Quindi $O(n)$.

ATTENZIONE: non è sufficiente dire che il costo è quello di una DFS senza specificare che il lavoro in ogni nodo è costante. Quanto si lavora in ogni nodo durante la DFS dipende dal problema e dalla soluzione. Nel caso in cui in ogni nodo si lavorasse per $O(n)$, per esempio, il costo sarebbe $O(n^2)$ e sarebbe sbagliato affermare che sia $O(n)$.

Versione 2

Il livello del nodo può essere calcolato anche dal basso verso l'alto, senza la necessità di utilizzare il parametro k nella funzione ricorisva. La funzione prende in input un parametro t' che è il puntatore ad un nodo dell'albero T e il parametro di input al problema t.

- **Caso base:** (1) $t' = t$, in questo caso restituiamo zero, (2) $t' \neq t$ e t' è una foglia: significa che non abbiamo trovato il nodo e restituiamo -1.
- **Caso ricorsivo:** $t' \neq t$ e t' non è una foglia. Continuiamo a cercare nei sottoalberi non vuoti. Il risultato delle chiamate può essere -1 (il nodo non è in quel sottoalbero), oppure un valore ≥ 0. Nel caso in cui entrambe le chiamate restituiscano -1, si ritorna -1 (il nodo non si trova nel sottoalbero radicato in t), altrimenti si torna 1 + il valore non negativo.

Aggiungiamo una funzione principale che controlla se T e/o il puntatore t sono nulli. Se lo sono, allora restituisce -1, altrimenti restituisce il livello di t invocando la funzione descritta precedentemente con parametri opportuni.

```
LivelloNodo(T,t)
  if T = NIL OR t = NIL then return -1
  return livello(T,t)


livello(t',t)
  if t' = t then return 0
  if t' ≠ t AND t'.left = NIL AND t'.right = NIL then return -1
  dx := -1
  sx := -1
  if t'.left ≠ NIL then sx := livello(t'.left,t)
  if sx ≥ 0 then return sx +1
            // il nodo si trova nel sottoalbero sinistro, non e' necessario continuare a cercare a destra
  if t'.right ≠ NIL then dx := livello(t'.right,t)
  if dx ≥ 0 then return dx + 1// il nodo si trova nel sottoalbero destro
  return -1 // il nodo non si trova nell'albero radicato in t'
```

Osserviamo che la funzione non viene mai invocata su un sottoalbero vuoto.

Per il costo computazionale valgono le stesse osservazioni fatte nella versione precedente.

Per la parte da tre punti si può ragionare come nell'altra versione precedente, modificando opportunamente la chiamata alla funzione `livello`.

OSSERVAZIONE: per la domanda da $+3$ punti: si può, in entrambe le versioni, modificare la funzione `livello` per far ritornare una coppia: il livello del nodo t e il puntatore al padre. Questa soluzione non richiede di invocare a parte una seconda funzione per determinare se i nodi sono fratelli o meno. La scrittura dello pseudocodice viene lasciata per esercizio.

ERRORI PIÙ COMUNI:

- Nell'esempio: sbagliare a contare i livelli dell'albero: il livello della radice è zero e non uno
- Assumere che i nodi abbiano identificativi nell'intervallo [1..n]. I campi sono solo quelli indicati dalla traccia (non c'è un campo ID) e non si possono utilizzare le chiavi perché non è detto che queste siano uniche e nell'intervallo desiderato.
- Utilizzare la versione iterativa della BFS, la traccia chiedeva esplicitamente di progettare un algoritmo ricorsivo
- Assumere che l'albero sia un Binary Search Tree e scendere nel sottoalbero destro o sinistro con un test sulla chiave del nodo radice
- Assumere che l'albero sia un Heap e possa essere implicitamente implementato in un array
- Nella spiegazione a parole: (1) non indicare esplicitamente il caso base e ricorsivo, oppure non indicare cosa fare nei due casi (2) non spiegare come gestire il parametro aggiuntivo (rispetto alla traccia) che indica l'altezza del nodo
- Sbagliare ad usare un contatore globale per l'altezza del nodo
- Solito problema di utilizzare le funzioni ricorsivamente ma non salvare da nessuna parte il risultato delle chiamate. Vedere vecchie soluzioni per spiegazioni più dettagliate.

- Invocare funzioni ricorsive su figli vuoti senza aver considerato il caso base dell'albero vuoto.
- Sommare i risultati delle chiamate ricorsive dei due sottoalberi nel caso in cui da uno dei due possa ritornare un valore diverso da zero.

Algoritmi e Strutture Dati **Anno Accademico 2021–2022**

Quarto Appello - 15 Settembre 2022

SECONDA PARTE (la traccia non deve essere consegnata)

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 9 punti)

Si consideri il seguente problema:

INPUT: Una coda Q che contiene valori interi ordinati in ordine non crescente (in testa il massimo, il minimo in coda).
OUTPUT: Una coda Q' che contiene gli stessi valori di Q ma ordinati in ordine non decrescente (in testa il minimo, il massimo in coda).

1. Fornire una spiegazione a parole di un algoritmo che risolva il problema.
2. Fornire lo pseudocodice dell'algoritmo descritto a parole precedentemente utilizzando le code ad alto livello (ovvero utilizzando le primitive standard delle code, senza avere la possiblità di conoscerne la realizzazione a basso livello).
3. Studiare il costo computazionale dell'algoritmo proposto.

Promemoria primitive delle code:
`new_queue() --> Q`, `is_empty_queue(Q) --> True/False`, `enqueue(Q,val)`, `dequeue(Q) --> val`.

Esercizio 2. (max 12 punti + 3 extra punti assegnati solo se si è risposto ai punti 1,2 e 4)

Dato un albero binario $T = (V, E)$ e un nodo $v \in V$, si definisce *imbalance(v)* la differenza in valore assoluto tra il numero di foglie nel sottoalbero sinistro di v e il numero di foglie nel sottoalbero destro di v. Se v è una foglia, allora si definisce *imbalance(v)=0*. Infine, l'*imbalance(T)* dell'albero T è il massimo delle *imbalance* dei nodi in T, ovvero *imbalance(T)* $= \max_{v \in V}$ *imbalance(v)*.

Si consideri il seguente problema:

INPUT: Un albero binario e **completo** T realizzato con nodi e puntatori (campi $val, left$ e $right$ con significato standard) .
OUTPUT: L'*imbalance(T)*.

1. Mostrare un esempio di input e output del problema, con un albero che abbia almeno 8 nodi.
2. Fornire una spiegazione a parole di un algoritmo che risolva il problema.
3. Fornire lo pseudocodice dell'algoritmo descritto a parole precedentemente.
4. Studiare il costo computazionale dell'algoritmo proposto.

5. **Extra (+ 1 punto):** Mostrare un'istanza del problema per cui vale che *imbalance(*T*)* = *imbalance(*v*)* e v **NON** è la radice dell'albero.

6. **Extra (+ 2 punti):** Dato un albero binario e completo T con f foglie, qual'è il massimo valore che può assumere *imbalance(*T*)*? Fornire una spiegazione e un esempio di istanza in cui *imbalance(*T*)* è massima.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)
Dare una definizione di DAG. Spiegare perché in ogni DAG esistono almeno un pozzo e una sorgente. Spiegare cosa si intende per linearizzazione (o ordinamento topologico) di un DAG e spiegare perché è sempre possibile linearizzare un DAG.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Versione 1.

Si utilizza una struttura dati DINAMICA di appoggio. Per invertire l'ordine degli elementi abbiamo bisogno di una struttura che non lavori con la logica FIFO come la coda, ma possiamo utilizzare una pila che, invece, utilizza la logia LIFO.

L'algoritmo lavora in due passi:

1. Svuota la coda nella pila: in questo modo l'elemento in testa alla coda diventa quello più lontano dalla testa della pila.
2. Svuota la pila in una nuova coda: in questo modo il primo elemento ad uscire dalla pila è l'ultimo che vi è entrato (ovvero il più piccolo) e l'ultimo ad uscire dalla pila è il primo che è entrato (ovvero il più grande).

```
InvertiCoda(Q)
    Q':= new_queue()
    S:= new_stack()
    while NOT is_empty_queue(Q) do
       push(S,dequeue(Q))
    while NOT is_empty_stack(S) do
       enqueue(Q',pop(S))
    return Q'
```

Al posto della pila si può utilizzare una lista, inserendo in testa e poi estraendo dalla testa (che è esattamente il comportamento della pila, non è un caso se la lista viene utilizzata per implementare la coda).

Osservazione 1: **usare array statico** in cui andare a salvare gli elementi della coda per poi inserirli in ordine inverso in un'altra coda **non è una soluzione molto ragionevole** (anche se può funzionare) e introduce l'inutile complicazione di dover contare gli elementi della coda per poter dimensionare l'array nel modo corretto. Per contare gli elementi, infatti, è necessario svuotare la coda Q spostando i suoi elementi in un'altra coda di appoggio Q_{tmp} (si potrebbe utilizzare la stessa coda, reinserendo gli elementi in coda dopo averli estratti per contarli, solo nel caso in cui non ci sono elementi uguali. In caso contrario, infatti, non si riesce a stabilire quando abbiamo finito di contare.). Poi si deve svuotare la coda di appoggio e riempire l'array e, infine, copiare gli elementi dell'array nella nuova Q' che verrà resitutita come output.

Il problema è che si cerca di utilizzare una struttura (l'array) che NON si presta a memorizzare sequenze lineari dinamiche per gestire una sequenza lineare dinamica. Vista la disponibiltà di altre strutture dinamiche, è più opportuno utilizzare quelle. Evitare complicazioni significa ridurre la proabilità di commettere errori.

Osservazione 2: si potrebbe pensare di ovviare la problema di contare gli elementi di Q utilizzando un **array dinamico**. La soluzione funziona, perché l'array dinamico si ridimensiona nel caso in cui lo spazio attualmente allocato per l'array non sia sufficiente. Il rovescio della medaglia è che ridimensionare l'array non ha un costo trascurabile. Quello che succede è che lo spazio allocato all'array viene duplicato, ma difficilmente lo spazio aggiuntivo si troverà in memoria in coda allo spazio attualmente allocato per l'array, quindi il contenuto dell'array deve essere copiato nella nuova locazione dell'array ridimensionato. **Il costo, nel caso peggiore, di un inserimento nell'array** diventa quindi $O(n)$ (dove n

sono il numero degli elementi già nell'array) e **non è più costante** (perché si devono ricopiare gli n elementi prima di poter inserire quello nuovo). Se vogliamo parlare di costi costanti, dobbiamo prendere in considerazione il costo *ammortizzato* (di cui non abbiamo parlato a lezione) di una operazione di inserzione, ovvero il suo costo *in media* su una sequenza di operazioni di inserzioni.

Queste considerazioni sui costi devono essere riportate nella discussione del costo computazionale dell'algoritmo. In ogni caso, anche questa soluzione non è particolarmente ragionevole, né interessante.

Versione 2. Invece di utilizzare esplicitamente una pila, si può farne un **uso implicito** utilizzando la ricorsione e il suo stack dei record di attivazione. La soluzione funziona ed è interessante come esempio di utilizzo della ricosione, ma dal punto di vista pratico è meno efficiente perché si introduce l'overhead della ricorsione.

Scriviamo quindi una procedura ricorsiva che prende in input una coda q e manipola una coda globale Q', inizialmente vuota, in modo che alla fine Q' contenga gli elementi di q in ordine inverso:

- **Caso base:** la cosa q è vuota, non c'è niente da fare;
- **Caso ricorsivo:** la coda q non è vuota. Si estrae il primo elemento di q e lo si memorizza in una variabile locale, si inverte ricorsivamente quello che resta di q in Q' e, solo a questo punto, si inserisce in coda a Q' l'elemento precedentemente salvato in u.

```
InvertiCoda(Q)
  Q' := new_queue() // variabile globale
  Inverti(Q)
  return Q


Inverti(q)
  if NOT is_empty_queue(q)
     then
       u := dequeue(q)
       Inverti(q)
       enqueue(Q',u)
```

ESERCIZIO: provate ad eseguire il codice (a mano!) su una coda Q con quattro o cinque elementi.

ERRORI PIÙ COMUNI:

- Estrarre dalla coda un elemento alla volta e inserirli in un'altra coda nello stesso ordine con cui escono dalla prima. Provate, la coda non si inverte.
- Cercare ripetutamente il minimo nella coda di input e inserirlo nella coda nuova. Questo è un modo di procedere che, se implementato correttamente, porta al risultato corretto , ma è inefficiente, contorto, e non sfrutta per niente le proprietà della coda: la sua dinamicità e la logica FIFO. Infatti, quasi tutti coloro che hanno provato a risolvere il problema in questo modo hanno commesso errori importanti.
- Agire sulla coda a basso livello (ovvero manipolare i puntatori). La traccia dice esplicitamente di non farlo e nessuno dà la garanzia che la coda sia effettivamente implementata con una lista.

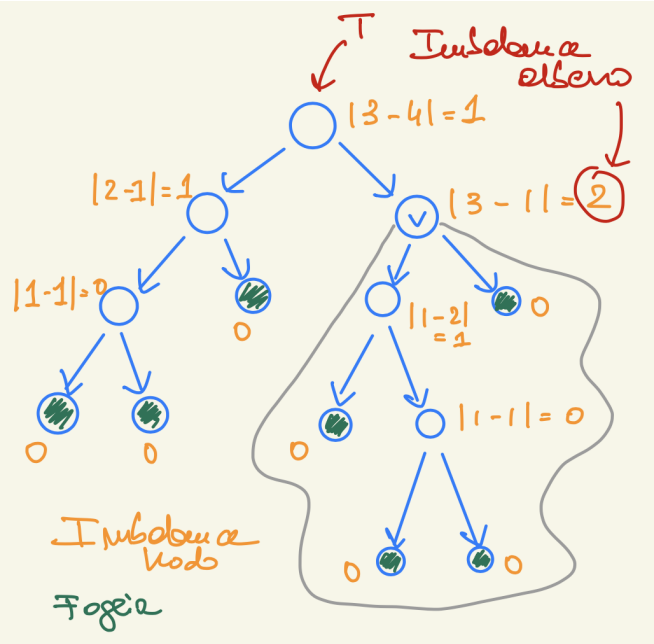

Figura 3: Esempio per il punto (1): tutto l'albero. In arancione le imbalance dei singoli nodi, in rosso l'imbalance di tutto l'albero (uguale al valore massimo dei valori in arancione). Esempio per il punto (5): l'albero ha imbalance 2 che è l'imbalance massima dei nodi, e che non coincide con l'imbalance della radice. Esempio per il punto (6): solo il l'albero radicato in v evidenziato in girgio.

SOLUZIONE Esercizio 2.

Punto 1 - Esempio. La figura **??** mostra un esempio di albero binario completo e dell'imbalance dell'albero. In arancione sono indicate le imbalance dei singoli nodi: zero (per definizione) per le foglie, il valore assoluto della differenza tra il numero delle foglie del sottoalbero sinistro e il numero delle foglie del sottoalbero destro per i nodi interni. In rosso è evidenziato il massimo tra le imbalance dei nodi, che è l'imbalance dell'albero.

Punti 2, 3 e 4. Per trovare la soluzione al problema è necessario calcolare l'imbalance di ogni nodo (radice inclusa) e calcolare il massimo tra questi valori. Per calcolare l'imbalance di un nodo è necessario conoscere il numero di foglie del sottoalbero sinistro e del sottoalbero destro. Per calcolare l'imbalance di un albero si può sfruttare la sua natura ricorsiva: l'imbalance di un albero t è il massimo tra l'imbalance della radice, l'imbalance del sottoalbero destro e l'imbalance del sottoalbero sinistro. Questo perché l'imbalance di un sottoalbero è già il massimo tra le imbalance dei nodi nel sottoalbero (e non della radice). L'imbalance dei sottoalberi si calcola ricorsivamente.

Il fatto di avere in input un albero completo ci permette di evitare di dover trattare in modo diverso i figli di un nodo: o ci sono entrambi, o non ce ne è neanche uno.

Versione 1

Con una visita DFS, calcoliamo contemporaneamente il numero di foglie nei due sottoalberi e le imbalance dei due sottoalberi. Il lavoro da fare in ogni nodo durante la visita è quello di calcolare l'imbalance del nodo radice e restituire l'imbalance e il numero di foglie dell'albero radicato nel nodo.

La funzione `Imbalance_tree` prende in input un albero t (binario e completo) e restituisce una coppia di valori: il primo valore è il numero di foglie di t, il secondo l'imbalance di t (NON della radice, ma il massimo tra le imbalance di tutti i nodi, radice compresa, nel sottoalbero).

- **Caso base:** L'albero è una foglia. Si restituisce la coppia $(1, 0)$, ovvero l'albero ha una foglia e ha imbalance 0, per definizione.
- **Caso ricorsivo:** L'albero non è una foglia, allora ha due figli (perché è completo). Si chiama ricorsivamente la funzione sui due figli, si ottengono due coppie (f_s, i_s) e (f_d, i_d) che contengono il numero di foglie del sottoalbero sinistro, rispettivamente destro, e l'imbalance del sottoalbero sinistro, rispettivamente destro. A questo punto ci sono tutti i dati necessari per calcolare l'output della chiamata:
  - Il numero di foglie dell'albero t non è altro che la somma delle foglie dei due sottoalberi, ovvero $f_s + f_d$;
  - L'imbalance di t è il massimo tra l'imbalance della radice di t, l'imbalance del sottoalbero sinistro e l'imbalance del sottoalbero destro. Gli ultimi due valori si trovano nei risultati delle chiamate ricorsive. Per calcolare l'imbalance della radice, invece, si usa la definizione di imbalance in un nodo: differenza, in valore assoluto, del numero di foglie dei due sottoalberi, ovvero $|f_s - f_d|$. Quindi, il secondo valore da restituire è $\max\{i_s, i_d, |f_s - f_d|\}$.

Introduciamo una funzione principale che prende in input l'albero T, controlla che non sia vuoto e invoca `Imbalance_tree`. Infine restituisce il valore dell'imbalance nel risultato della chiamata di `Imbalance_tree`.

```
Imbalance(T)
  if T ≠ NIL
     then
       (f,i) := Imbalance_tree(T)
       return i


Imbalance_tree(t)
  if foglia(t) then return (1,0)
  // i figli ci sono entrambi perche' l'albero e' completo
  (fs, is) := Imbalance_tree(t.left)
  (fd, id) := Imbalance_tree(t.right)
  return ((fs+fd), max{ is, id, |fs-fd| } )


foglia(t)
  if t.left = NIL
     then return TRUE
     else return FALSE
```

Osservazione: per determinare se il nodo è una foglia basta controllare un figlio solo perché l'albero è completo.

Osservazione: non è necessario controllare che l'albero t sia vuoto in `Imbalance_tree`. **Perché?**

Il **costo computazionale** di questa versione è $O(n)$, dove n è il numero di nodi dell'albero. Infatti, la soluzione è una visita in post-ordine in cui il lavoro fatto in ogni nodo (escludendo le chiamate ricorsive) richiede tempo costante $O(1)$.

Versione 2

Effettuiamo una visita DFS in cui la visita di un nodo consiste nel calcolare la sua imbalance chiamando una funzione che conta le foglie nel sottoalbero di destra e in quello di sinistra. Le chiamate ricorsive sui due sottoalberi restituiscono le imbalance dei sottoalberi. Il risultato della visita del nodo radice è l'imbalance dell'albero radicato nel nodo, calcolata come massimo tra le imbalance dei sottoalberi e la sua.

La funzione `Imbalance_tree` prende in input un albero t e restituisce l'imbalance di t. Utilizza una funzione ausiliaria `imbalance_nodo` che calcola l'imbalance del nodo su cui è chiamata e che, a sua volta, utilizza un'altra funzione ausiliaria che conta il numero di foglie in un albero dato.

- **Caso Base:** L'albero t è una foglia, si restituisce zero (che, per definizione, è l'imbalance di una foglia).
- **Caso Ricorsivo:** L'albero t non è una foglia, allora ha due figli. Si chiama ricorsivamente la funzione sui due sottoalberi e si ricevono, come risultato, le imbalance i_s e i_d del sottoalbero sinistro e, rispettivamente, destro. Si calcola l'imbalance $root$ della radice di t: si contano le foglie del sottoalbero sinistro, quelle del sottoalbero destro, si calcola la differenza in valore assoluto. Si restituisce il massimo tra questi tre valori (ovvero, $max\{i_s, i_d, root\}$).

Introduciamo una funzione principale che prende in input l'albero T, controlla che non sia vuoto e invoca `Imbalance_tree` e restituisce il valore restitutito dalla chiamata.

```
Imbalance(T)
  if T ≠ NIL
     then return Imbalance_tree(T)

Imbalance_tree(t)
  if foglia(t) then return 0
  is := Imbalance_tree(t.left)
  id := Imbalance_tree(t.right)
  root := imbalance_nodo(t)
  return max { root, is, id }

imbalance_nodo(t)
  sx := conta_foglie(t.left)
  dx := conta_foglie(t.right)
  return |dx - sx|


conta_foglie(t)
   if foglia(t) then return 1
   return conta_foglie(t.left) + conta_foglie(t.right)

foglia(t)
```

```
if t.left = NIL
   then return TRUE
   else return FALSE
```

Osservazione: Si può risolvere il problema con questo approccio anche utilizzando una BFS. Provate per **esercizio**.

Il **costo computazionale** di questa versione è $O(n^2)$, dove n è il numero di nodi dell'albero. Infatti, si effettua una visita dell'albero in cui il lavoro in ogni nodo NON è costante, perchè non lo è il costo della funzione che conta le foglie. Quindi il costo computazionale è dato dal numero dei nodi dell'albero per il costo $O(n)$ della funzione di conteggio delle foglie.

Punto 5. L'esempio della figura **??** è anche un esempio che risponde a questa domanda. Infatti l'imbalance della radice dell'albero T passato in input è 1, mentre l'imbalance dell'albero T è 2, data dal nodo v.

Punto 6. Se l'albero ha f foglie, l'imbalance massima è $f-2$. Infatti, l'imbalance è data da $|f_s - f_d|$ dove f_s e f_d sono il numero di foglie nel sottoalbero sinistro e destro, rispettivamente, di un nodo interno $v \in V$ dell'albero.

Poichè l'albero è completo, v deve avere due figli e quindi ognuno dei sottoalberi (destro e sinistro) deve avere almeno una foglia. Quindi, abbiamo che $f_s, f_d \geq 1$.

Per lo stesso motivo, nessuno dei due sottoalberoi di v può contenere tutte le foglie dell'albero radicato in v, almeno una deve stare dall'altra parte). Quindi, abbiamo che

$$f_s, f_d \leq \text{ numero di foglie del sottoalbero radicato in } v < f.$$

Ovvero, $f_s, f_d \leq f-1$.

Supponiamo, senza perdita di generalità, che $f_s \geq f_d$ (in caso contrario si possono fare considerazioni duali). In questo modo possiamo eliminare il valore assoluto e abbiamo che $|f_s - f_d| = f_s - f_d$.

Mettendo tutto insieme:

$$Imbalance(T) = |f_s - f_d| = f_s - f_d \leq (f-1) - 1 = f - 2.$$

Vale l'uguaglianza se riusciamo a trovare un esempio di albero in cui imbalance vale esattamente $f-2$. Da quello che abbiamo detto prima, tale situazione avviene quando un sottoalbero ha il massimo numero di foglie possibili, e l'altro il minimo. Quindi un albero (binario e completo) in cui un sottoalbero ha una sola foglia, l'altro tutte.

Un esempio è l'albero radicato in v nella figura. Ha 4 foglie, una sta nel sottoalbero destro (e l'unico modo possibile affinché ce ne sia solo una è che il sottoalbero sia una foglia) e tutte le altre nel sottoalbero sinistro. L'imbalance dell'albero è quella della radice v, ovvero 2 (= 4 - 2).

ERRORI PIÙ COMUNI:

- Fornire un esempio con albero non completo. Sbagliare a calcolare le imbalance dei nodi e dell'albero.
- Calcolare imbalance solo della radice e restituire quello come risultato.
- Calcolare l'imbalance di un nodo come differenza o somma delle imbalance dei figli. È sbagliato e per farlo vedere basta mostrare un esempio in cui le affermazioni sono false. Si consideri di nuovo la figura con l'esempio: (1) la radice dell'albero T ha imbalance 1, i suoi figli (il nodo v e suo fratello) hanno imbalance 2 e 1, rispettivamente.

In questo caso, la somma delle imbalance dei figli è 3, che è diverso da 1. Quindi non si può calcolare l'imbalance di un nodo come la somma delle imbalance dei nodi figli; (2) si consideri il nodo v, ha imbalance uguale a 2. I suoi figli hanno imbalance zero e uno e la loro differenza in valore assoluto è uno, che è diverso da 2. Quindi non si può calcolare l'imbalance di un nodo come differenza, in valore assoluto, delle imbalance dei figli.

- Utilizzare in modo errato una variabile globale per ricordare il massimo delle imbalance durante una visita in cui si contano le foglie dei sottoalberi.
- Calcolare l'imbalance come differenza del numero dei nodi nei sottoalberi e non delle foglie.
- Utilizzare la notazione ad alto livello invece che i puntatori.

Algoritmi e Strutture Dati **Anno Accademico 2021–2022**

Quinto Appello
19 Gennaio 2023

SECONDA PARTE (la traccia non deve essere consegnata)

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 10 punti)

Si consideri il seguente problema:

INPUT: Un albero binario T realizzato con nodi e puntatori (campi $val, left$ e $right$ con significato standard, ma senza puntatore al padre).
OUTPUT: VERO se l'albero T è completo, FALSO altrimenti.

1. Fornire una spiegazione a parole di un algoritmo ricorsivo per risolvere il problema. Indicare esplicitamente il caso base e il caso ricorsivo e quali azioni intraprendere in entrambi i casi.
2. Fornire lo pseudocodice dell'algoritmo descritto a parole precedentemente.
3. Studiare il costo computazionale dell'algoritmo proposto. Fornire un esempio motivato di caso peggiore e uno di caso migliore.
4. Se si volesse testare che T sia anche perfettamente bilanciato, come si potrebbe procedere? Spiegare a parole (anche per punti, ma senza nascondere in frasi troppo generiche passaggi che richiedono operazioni non elementari).

Esercizio 2. (max 11 punti)

Si consideri una struttura dati astratta di nome `Pippo` con le seguenti caratteristiche:

- Memorizza un insieme di valori 0, 1 e 2, eventualmente ripetuti (cioè ogni valore può comparire più volte nella sequenza).
- Ha le seguenti primitive:
  - `new_Pippo(n)`: restituisce una nuova istanza di `Pippo` che può contenere al massimo n valori;
  - `insert(Pippo, x)`: modifica `Pippo` inserendo il valore `x`, se possibile;
  - `delete(Pippo, x)`: modifica `Pippo` eliminando una occorrenza del valore `x`, se esiste;
  - `count(Pippo, x)`: restituisce il numero di occorrenze del valore `x` in `Pippo`;

Si dia un'implementazione di `Pippo` e delle sue primitive (con una breve spiegazione) che soddisfi i seguenti vincoli:

- I valori sono memorizzati in un array;
- Le primitive hanno tutte costo costante $O(1)$.

Dare l'implementazione della struttura dati significa definire come sono organizzati i dati e le eventuali altre informazioni necessarie.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)
Spiegare come funziona la BFS su grafi, introducendo il problema approcciato dalla BFS, scrivendo e commentando lo pseudocodice. Quando e perché la BFS può essere utilizzata per calcolare l'albero dei cammini minimi da sorgente singola?

SOLUZIONI SECONDA PARTE

Purtroppo non ho avuto tempo di finire di scrivere le soluzioni in modo dettagliate o stilisticamente curate, ma volevo darvi un'idea delle soluzioni prima del prossimo appello. Qua trovate le idee, appena ho tempo completo le soluzioni con quello che manca.

SOLUZIONE Esercizio 1.

Un albero binario è completo se ogni nodo ha zero o due figli. Quindi, per risolvere l'esercizio, si può utilizzare la DFS e controllare in ogni nodo il numero di figli. A seconda dei casi si procede ricorsivamente o no.

Scriviamo una funzione ricorsiva `Completo(t)` che prende in input il puntatore alla radice di un albero t e restituisce un valore booleano vero o falso.

Casi base: l'albero t è una foglia, si restituisce `vero`. La radice di t ha solo un figlio, si restituisce `falso`.

Caso ricorsivo: la radice dell'albero t ha esattamente due figli, si procede ricorsivamente chiamando la funzione su entrambi i figli e restituendo l'AND dei risultati delle chiamate ricorsive.

Osservazioni: la ricorsione può terminare anche in un nodo interno che non sia una foglia. È fondamentale utilizzare i valori delle chiamate ricorsive per decidere se l'albero è completo, non basta controllare il singolo nodo. Non è specificato cosa fare se l'albero è vuoto, quindi è necessario fare in modo che non ci si torvi mai in quella situazione.

Aggiungiamo una funzione principale per controllare che T sia diverso dall'albero vuoto (e in questo caso si restituisce vero), e che invoca la funzione ricorsiva.

```
IsCompleto(T)
  if T = NIL
     then return vero
  return Completo(T)

Completo(t)
  if t.left = t.right = NIL
     then return vero
  if t.left = NIL OR t.right = NIL // non possono essere entrambi NIL
     then return falso
  return Completo(t.right) AND Completo(t.left) // i due sottoalberi non sono vuoti
```

Il costo computazionale dell'algoritmo è $O(n)$ perché, nel caso peggiore, si visitano tutti i nodi dell'albero, ognuno esattamente una volta sola, e il costo della visita di ogni nodo è costante, $O(1)$. Quindi $O(n) \cdot O(1) \in O(n)$.

Il caso **peggiore** è quello in cui è necessario visitare tutti i nodi e questo succede, per esempio, quando l'albero è completo. Infatti, solo dopo aver visitato tutti i nodi posso essere sicuro che ognuno di loro abbia zero o due figli. Se decidessi di non visitare un nodo, quello potrebbe avere un figlio solo e non lo scoprirei mai.

il caso **migliore** è quello in cui eseguo il minimo numero di operazioni, che in questo caso è costante $O(1)$. Si verifica quando T è nullo, oppure è una foglia (se analizzo la funzione ricorsiva).

Il problema di determinare e l'albero è perfettamente bilanciato si può risolvere in modi diversi. Ecco un paio di idee, provate a scrivere lo pseudocodice.

- Usiamo la DFS: calcoliamo l'altezza h dell'albero con la DFS come visto a lezione. Scriviamo una nuova DFS che prende in input un albero t, h, e un valore ℓ che indica il livello in T a cui si trova il nodo radice di t, e restituisce vero se le foglie si trovano tutte ad altezza h. Caso base: l'albero è una foglia, si restituisce il risultato del test $h = t$. Caso ricorsivo: si invoca la funzione sui due figli (se so già che l'albero è completo, altrimenti devo invocare sul figlio esistente), con h invariato e $\ell + 1$, e si restituisce l'AND delle chiamate ricorsive. La chiamata principale ha come valore dei parametri: T, h e 0.

- Usiamo una BFS. Se l'albero è perfettamente bilanciato, le foglie si trovano tutte allo stesso livello. Quindi nella coda della BFS si troveranno tutte una dopo l'altra e dopo di loro in coda non ci deve essere più niente. Per fare questo controllo è necessario avere un modo di capire, nella coda, quando finisce un livello e ne inizia un altro. Un modo per farlo è memorizzare in coda una coppia di valori invece che un valore solo. Nella coppia i due valori sono un nodo (come al solito) e il suo livello nell'albero. È necessario anche utilizzare una variabile in cui tenere traccia del livello corrente. In confronto di questa variabile con il livello del nodo estratto dalla coda ci dice se siamo ancora esaminando lo stesso livello oppure siamo passati al successivo. Se si trova una foglia "nel mezzo" di un livello (ovvero i nodi esaminati prima dello stesso livello non erano foglie), allora ci si può fermare perché l'albero non può essere perfettamente bilanciato. Se si trova una foglia come primo nodo di un nuovo livello, allora tutti gli altri nodi ancora in coda devono essere foglie. Se così è, si restituisce vero, altrimenti si restituisce falso.

ERRORI PIÙ COMUNI:

- Sbagliare la definizione di albero completo e risolvere un problema diverso
- Non salvare e/o non utilizzare i valori delle chiamate ricorisve, ma restituire vero o falso solo a seconda dello stato del nodo che si analizza (ovvero se ha zero, uno o due figli)
- Dichiarare che il costo computazionale sia $O(n)$ perché "è come una DFS", senza specificare che il lavoro in ogni nodo è costante.
- Per il punto 4, un albero è perfettamente bilanciato se tutte le sue foglie si trovano allo stesso livello. Non sono vere le seguenti affermazioni:
  - T è perfettamente bilanciato se per ogni nodo il sottoalbero di destra ha lo stesso numero di nodi del sottoalbero di sinistra (neanche se ho già dimostrato che T è completo)
  - T è perfettamente bilanciato se la topologia del sottoalbero di destra è uguale alla topologia del sottoalbero di sinistra (neanche se ho già dimostrato che T è completo)

  provate a produrre un controesempio per ciascuna affermazione.

SOLUZIONE Esercizio 2.

Versione 1 Memorizzare i valori nell'array in modo da avere prima tutti gli zero, poi a seguire gli uni e a seguire ancora i due. Per poter gestire le primitive in tempo costante è necessario tenere traccia dell'ultima posizione dell'array in cui sono memorizzati i diversi valori. Quindi la nostra struttura dati deve avere un array A per memorizzare i valori e tre indici, che memorizziamo in un altro array di tre elementi, $Index$. Infine, il numero di occorrenze dei singoli valori si può calcolare per differenza degli indici in $Index$, ma è più comodo tenere il conto a parte (si evitano un certo numero di controlli, non difficili ma che servono per tenere conto della possibiltà che alcuni valori siano assenti). A questo scopo utilizziamo un altro array $Count$. Un'istanza di Pippo sarà un puntatore ad un nodo con tre campi, che puntano ai tre array.

Per fare un'inserzione lavoriamo nel modo seguente: aggiungiamo il valore da inserire in coda agli altri identici ad esso. Nel caso in cui questa cella fosse occupata da un un'occorrenza di un valore maggiore di quello da inserire, si sposta il valore in coda a quelli identici ad esso e si ripete ancora lo scambio se necessario. Per la cancellazione, si procede in modo analogo ma duale. Per il conteggio basta restituire il valore opportuno in $Count$ (che ovviamente deve essere tenuto aggiornato nelle operazioni di inserzione e cancellazione).

Appena ho tempo aggiungo lo pseudo codice e miglioro la spiegazione

Versione 2

Si possono memorizzare i valori nella sequenza mantenendoli "disordinati". Si aggiungono delle strutture ausiliari per tenere traccia degli indici utilizzati e quelli liberi.

Appena ho tempo finisco di scrivere la soluzione

ERRORI PIÙ COMUNI:

- Non memorizzare gli elementi dell'insieme. È vero che non vengono mai restituititi, ma la traccia diceva esplicitamente che i valori dovevano essere memorizzati in un array, non che dovevano essere contati.
- Memorizzare gli elementi in un array, ma utilizzare tempo non costante, ma lineare, per le primitive.

Algoritmi e Strutture Dati **Anno Accademico 2021–2022**

Sesto Appello
15 Febbraio 2023

SECONDA PARTE (la traccia non deve essere consegnata)

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 10 punti)

Si consideri il seguente problema:

INPUT: un array A contenente n valori interi ordinati in senso non decrescente (possono essere presenti valori duplicati) e un intero x.
OUTPUT: l'indice dell'ultima (più a destra) occorrenza di x in A, oppure -1 se il valore x non è presente in A.

Progettare un algoritmo basato sulla tecnica **Divide et Impera** che risolva il problema e:

- Descrivere l'algoritmo a parole specificando espressamente le azioni da compiere nelle fasi Divide, Impera e Combina.
- Mostrare un esempio con un array A contentente almeno 7 valori e in cui l'intero cercato sia ripetuto più volte in A.
- Scrivere lo pseudocodice dell'algoritmo presentato.
- Analizzare il costo computazionale dell'algoritmo.

Esercizio 2. (max 11 punti + 2 punti)

Un *grafo con nodi pensati* è un grafo non orientato $G = (V, E)$ in cui a ciascun nodo $v \in V$ è associato un peso intero $w(v)$ (che può essere positivo o negativo). Un cammino $< v_1, v_2, \ldots, v_k >$ si dice *monotono* se $v_i \in V$ per ogni $i \in [1..k]$ e vale che $w(v_1) < w(v_2) < \ldots < w(v_k)$ (ovvero è composto da nodi del grafo che, attraversati nell'ordine del cammino, hanno pesi strettamente crescenti).

Il *problema del cammino monotono* è il seguente:

INPUT: un grafo con nodi pesati $G = (V, E)$, tale che $V = \{1, 2, \ldots, n\}$, funzione di peso sui nodi $w : V \to Z$, un nodo sorgente $s \in V$ e un nodo destinazione $d \in V$.
OUTPUT: stampa dei nodi su un cammino monotono che inizia in s e termina in d se esiste, oppure "il cammino monotono non esiste" in caso contrario. I nodi possono essere stampati in entrambe le direzioni (da s a d, oppure da d a s).

- Si dia un esempio di grafo con nodi pesati che contenga almeno un cammino monotono, in cui il grafo abbia almeno 7 nodi e il cammino almeno 3.
- Descrivere a parole e fornire lo pseudocodice di un algoritmo per il problema del cammino monotono.
- Discutere il costo computazionale dell'algoritmo proposto.
- (Bonus + 2) Dimostrare che un cammino monotono in un grafo pesato è aciclico.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)
Definire la struttura dati *Heap* nella sua versione "originale" ad albero, e spiegare come questa possa essere implementata utilizzando un array. Scegliere una delle primitive della struttura dati Heap e spiegare come possa essere implementata.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Il problema si può risolvere in molti modi diversi, alcuni simili alla ricerca binaria, l'altri alla scansione di un array, altri ancora simili al mergesort. In ogni caso, dovendo utilizzare la tecnica *Divide & Impera* la cosa importante è che il problema di una certa dimensione si risolva utilizzando soluzioni **allo stesso problema** su input di dimensione più piccola di quello originario. Quindi, è importante specificare il problema che si vuole affrontare, dicendo esattamente come è fatto l'input e cosa ci si aspetta in output, che può essere leggermente diverso da quello della traccia. Se questo è il caso, si introdurrà una funzione principale per risolvere esattamente il problema della traccia.

Ovviamente viene naturale utilizzare la ricosione e la funzione principale non sarà altro che la chiamata principale che ci permette di definire il valore dei parametri per risolvere il problema originario della traccia.

Non essendoci vincoli relativi al tempo di esecuzione nella traccia, tutte le soluzioni andavano bene. L'importante era riuscire a calcolarlo correttamente.

Vi ricordo che, come scritto nella traccia, anche se la tecnica si chiama solo *Divide & Impera*, c'è anche la terza fase, *Combina*.

Vi propongo tre soluzioni che adottano approcci diversi.

Versione 1. La binary search che abbiamo visto a lezione non funziona per risolvere questo problema, perché si ferma alla prima occorrenza trovata. Possiamo però modificarla in modo che continui a cercare il valore x anche dopo averlo già trovato. L'importante è stabilire come fare a continuare a cercare o fermarsi.

Problema: data una porzione di array $A[i..j]$ e un valore x, restituire l'indice dell'occorrenza di x più a destra in $A[i..j]$ se esiste, altrimenti restituire -1.

- *Divide:* calcola l'indice k dell'elemento centrale di $A[i..j]$, se x è minore di $A[k]$ considera la metà di sinistra dell'array escludendo l'elemento centrale (ovvero $A[i..k-1]$), altrimenti considera la metà di destra includendo l'elemento centrale (ovvero $A[k..j]$)
- *Impera:* risolvi il sottoproblema sulla metà considerata, ottenendo come risultato il valore k_1
- *Combina:* niente, solo restituire il valore k_1

Il *caso base*, ovvero il caso in cui il problema è facile da risolvere e non è necessario procedere a "dividere" ancora, si ha quando ci troviamo in una delle due seguenti situazioni: (1) $i = j$ e $A[i] = x$, e si restituisce i; (2) $i > j$, e si restituisce -1.

```
UltimaOccorrenza1(A, i, j, x)
  if i > j then return -1
  if i = j AND A[i] = x then return i
  k := ⌊ (i+j)/2 ⌋
  if A[k] ≤ x
     then return UltimaOccorrenza1(A,k,j,x)
     else return UltimaOccorrenza1(A,i,k-1,x)
```

Chiamata principale `UltimaOccorrenza1(A,0,n-1,x)`.

Il costo computazionale è lo stesso del caso peggiore della Binary Search, perché andiamo avanti sempre fino ad avere una porzione di array con un solo elemento o zero. Quindi $O(\log n)$.

Osservazione: la soluzione può essere modificata aggiungendo un caso base che ci permette di terminare prima in alcuni casi. Supponiamo di aver trovato un'occorrenza di x all'indice k: se all'indice $k+1$ troviamo ancora x, allora quella all'indice k non era l'occorrenza più a destra e si deve continuare a cercare; altrimenti all'indice k abbiamo trovato l'occorrenza di x che stavamo cercando. Attenzione, che per fare il controllo sull'elemento alla posizione $i+1$ è necessario che $i < n-1$.

Attenzione che questa soluzione non cambia il numero di chiamate ricorisve da fare nel caso peggiore, rispetto alla versione scritta nello pseudocodice. Provate a pensare ad un'istanza in cui x è presente nell'array ma la modifica non basta fermarsi prima.

Versione 2. Eseguiamo una scansione ricorsiva dell'array alla ricerca dell'elemento desiderato.

Problema: data una porzione di array $A[i..n-1]$ e un valore x, restituire l'indice dell'occorrenza di x più a destra in $A[i..j]$ se esiste, altrimenti restituire -1.

- *Divide:* considera un sottoproblema di dimensione più piccola in cui l'array "perde" la prima posizione, ovvero $A[i+1..n-1]$
- *Impera:* risolvi il problema sul sottoproblema e ottieni k come risultato
- *Combina:* niente, solo restituire il valore k

Il *caso base*, ovvero il caso in cui il problema è facile da risolvere e non è necessario procedere a "dividere" ancora, si ha quando ci troviamo in una delle due seguenti situazioni: (1) la porzione di array data in input ha dimensione nulla (ovvero $i > n-1$), e si restituisce -1; (2) la porzione di array ha dimensione uno e contiene proprio il valore x, si restituisce l'indice in cui compare questa occorrenza di x, ovvero $n-1$; (3) il primo elemento della porzione di array (ovvero $A[i]$) è esattamente x e il secondo è diverso da x, e si restituisce i.

```
UltimaOccorrenza2(A, i, n, x)
  if i = n-1 AND A[i] = x
     then return n-1
     else return -1
  if A[i] = x AND A[i+1] ≠ x
     then return i
  return UltimaOccorrenza2(A, i+1, n, x)
```

Chiamata principale `UltimaOccorrenza2(A,0,n,x)`.

Osserviamo che nel secondo `if` non è necessario controllare che $i+1 \leq n-1$, perché se abbiamo passato il primo test, sicuramente abbiamo che $i < n-1$.

Il costo computazionale è quello di una scansione lineare di A. Infatti, (1) per ogni invocazione di `UltimaOccorrenza2`, escludendo la chiamata ricorsiva, paghiamo un costo costante; (2) nel caso peggiore (in cui l'ultima occorrenza di x si trovi in posizione $n-1$ o non esista), vengono invocate n chiamate di `UltimaOccorrenza2` (inclusa la chiamata principale). In totale, il costo è pari al prodotto del costo di una chiamata e il numero di chiamate, ovvero $O(n)$.

Versione 3. Lavoriamo in modo analogo a `MergeSort`.

Problema: data una porzione di array $A[i..j]$ e un valore x, restituire l'indice dell'occorrenza di x più a destra in $A[i..j]$ se esiste, altrimenti restituire -1.

- *Divide:* dividi l'array in due metà di uguale dimensione (a meno di uno)
- *Impera:* risolvi il problema sulle due metà, ottenendo due valori k_1 e k_2 come soluzioni ai sottoproblemi.
- *Combina:* restituisci il massimo tra k_1 e k_2

Il *caso base*, ovvero il caso in cui il problema è facile da risolvere e non è necessario procedere a "dividere" ancora, si ha quando ci troviamo in una delle due seguenti situazioni: (1) $i = j$ e $A[i] = x$, e si restituisce i; (2) $i = j$ e $A[i] \neq x$, e si restituisce -1.

```
UltimaOccorrenza3(A, i, j, x)
  if i = j AND A[i] = x then return i
  if i = j AND A[i] ≠ x then return -1
```

```
k := ⌊ (i+j)/2 ⌋
return max { UltimaOccorrenza(A,i,k,x) , UltimaOccorrenza(A,k+1,j,x) }
```

Chiamata principale `UltimaOccorrenza3(A,0,n-1,x)`.

Anche il costo computazionale di questa soluzione è $O(n)$, ma è un pò più faticoso dimostrarlo rispetto alle versioni precedenti. Ogni invocazione di `UltimaOccorrenza3`, escludendo la chiamata ricorsiva, richiede anche qua un costo costante, ma bisogna contare il numero di invocazioni correttamente. Si può ragionare in modo simile a come si analizza `Mergesort` (ma attenzione che il conto finale è diverso perché in questo esercizio il costo della fase *Combina* è costante, mentre in `Mergesort` non lo è):

- Ci sarà UNA invocazione su un problema di dimensione n;
- ci saranno DUE invocazioni su un problema di dimensione $n/2$;
- ci saranno QUATTRO invocazioni su un problema di dimensione $n/4$;
- ... così via....
- ci saranno $n/4$ invocazioni su un problema di dimensione 4;
- ci saranno $n/2$ invocazioni su un problema di dimensione 2;
- ci saranno n invocazioni su un problema di dimensione 1.

Per fare i conti in modo più agevole (e anche perché è il caso peggiore), assumiamo che n sia una potenza di 2, ovvero che esista un k intero positivo per cui $n = 2^k$. Il numero di invocazioni è:

$$1 + 2 + 4 + \ldots + \frac{n}{4} + \ldots + \frac{n}{2} + n = \frac{n}{2^k} + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \ldots + \frac{n}{2^1} + \frac{n}{2^0} = n \sum_{i=0}^{\log n} \frac{1}{2^i} \leq n \sum_{i=0}^{+\infty} \frac{1}{2^i} \leq 2n \in O(n),$$

per le proprietà della serie geometrica.

ERRORI PIÙ COMUNI:

- Non descrivere o sbagliare la descrizione delle fasi di *Divide & Impera*, o dimenticarsi qualche fase
- Sbagliare a studiare il costo computazionale (sopratutto in varianti della versione3)
- Combinare i risultati delle chiamate ricorsive nel modo sbagliato
- Dimenticare il `return` prima delle chiamate ricorsive (con che valori lavora la funzione chiamante se non riceve niente in ritorno dalle chiamate ricorsive?)
- Non risolvere correttamente il problema

SOLUZIONE Esercizio 2.

Il problema si può risolvere con una visita BFS o una visita DFS. L'idea di fondo è la stessa in entrambi i casi: durante l'esplorazione degli archi incidenti in un nodo, si prosegue la visita seguendo quelli che portano a nodi non ancora visitati E che abbiano un peso maggiore del nodo che stiamo visitando. La visita parte dal nodo sorgente e, se riesce ad arrivare al nodo destinazione, allora esiste un cammino monotono dalla sorgente alla destinazione, altrimenti non esiste. Per poter stampare i nodi sul cammino si può utilizzare il vettore dei padri risultante dalla visita, oppure con la DFS al ritorno dalla ricorsione.

L'idea funziona perché equivale a lavorare su un grafo diretto G' in cui gli archi sono gli stessi del grafo di partenza G e gli archi sono diretti dal nodo con penso più piccolo al quello più alto (non c'è arco se due nodi hanno lo stesso peso). Se su questo grafo diretto G' esiste un cammino da s a d, allora con una visita verrà trovato e sarà un cammino monotono sul grafo di partenza G.

Se invece il cammino monotono non esiste in G, non può esistere un cammino diretto da s a d in G' e quindi la visita fallirà nel cercarlo.

Versione 1. Utilizziamo una DFS con sorgente s (che non riparte da altri nodi eventualmente non visitati) e il vettore dei padri che viene utilizzato anche per controllare se un nodo è già stato visitato. Il vettore dei padri viene inizalizzato, per ogni nodo, a -1 per indicare che non gli è ancora stato assegnato nessun padre dalla visita. Quando la visita incontra un nodo per la prima volta gli assegna come padre il nodo che ha permesso di scoprirlo e il padre avrà sicuramente un identificativo che è diverso da -1. Quindi possiamo decidere se un nodo v sia stato visitato o meno utilizzando il valore memorizzato nel vettore dei padri all'indice v: se è uguale a -1 non è stato visitato, altrimenti lo è stato. Ovviamente la stessa cosa è vera per la destinazione d: se alla fine della visita non è stato raggiunto, nel vettore dei padri all'indice d si troverà ancora -1.

```
CamminoMonotono(G,w,s,d)
  for i = 1 to n do
      parent[i] := -1
  Visita(G,w,s,d)
  if parent[d] = -1
     then print "non esiste il cammino"
     else Stampa(s,d)
```

```
Visita(G,w,v,d)
  for all (v,u) ∈ E do
      if parent[u] ≠ -1 AND w(u) > w(v) // il nodo u non e' stato visitato e ha un peso piu' grande
         then
           parent[u] := v
           if u = d then return // d e' stato raggiunto, e' inutile continuare con la ricorsione
           Visita(G,w,u,d)
```

La procedura di stampa dei nodi del cammino può essere implementata in due modi diversi (ma sempre utilizzando la ricorsione) a seconda che si voglia stampare i nodi da s a d o viceversa. Cambia solo l'ordine con cui si stampano i nodi.

```
Stampa(s,v) // da d a s
   print v
   if v ≠ s then Stampa(s, parent[v])
```

oppure

```
Stampa(s,v) // da s a d
   if v ≠ s then Stampa(s,parent[v])
   print v
```

ESERCIZIO. Modificare lo pseudo-codice in modo che tutte le chiamate ricorsive ancora aperte al momento della scoperta di d vengano chiuse senza esplorare altri archi.

Costo Computazionale: Il costo di `Visita` è lo stesso di una classica visita DFS (non sono state introdotte variazioni che modificano il costo asintotico) e la stampa richiede tempo addizionale lineare nel numero di nodi sul cammino da s a d, che nel peggiore dei casi può contenere tutti i nodi del grafo. Quindi abbiamo $O(|V| + |E|)$ per la visita, più $O(|V|)$ per la stampa. In conclusione abbiamo $O(|V| + |E|)$.

Versione 2. Modifichiamo la soluzione precedente in modo da integrare la stampa nella ricorsione della subroutine `Visita`. A questo scopo modifichiamo la visita in modo che, se arriva al nodo d, restituisca vero. Il vettore dei padri non è più necessario, ma ci serve l'array `visited` per evitare di tornare a visitare lo stesso nodo più volte.

```
CamminoMonotono(G,w,s,d)
  for i = 1 to n do
      visited[i] := False
  if NOT Visita(G,w,s,d)
     then print "non esiste il cammino"


Visita(G,w,v,d)
  visited[v] := True
  if v = d // caso base:  siamo arrivati in d, lo stampiamo e restituiamo vero
     then
       print v
       return True
  for all (v,u) ∈ E do // caso ricorsivo:  continuiamo la visita esplorando gli archi incidenti in v
      if visited[u] = False AND w(u) > w(v)
         then
           if Visita(G,w,u,d)
              then
                print v
                return True
  return False
```

Costo Computazionale: Il costo di `Visita` è lo stesso di una classica visita DFS (non sono state introdotte variazioni che modificano il costo asintotico), quindi in conclusione abbiamo $O(|V| + |E|)$.

Versione 3. Usare una BFS invece che una DFS usando la stessa idea della versione 1. ESERCIZIO.

Domanda Bonus: supponiamo che un cammino monotono da s a d contenga un ciclo e che questo ciclo sia formato dai nodi $\{v_1, v_2, \ldots, v_t\}$ in questo ordine (vuol dire che dopo v_i nel ciclo troviamo v_{i+1} e che dopo v_t troviamo v_1). Poiché è un cammino monotono deve essere

$$w(v_1) < w(v_2) < \ldots < w(v_t),$$

ovvero, seguendo il cammino, il nodo che viene dopo ha un peso più grande di quello che lo precede. Da questo si deduce che $w(v_1) < w(v_t)$.

Ma se abbiamo un ciclo, il nodo v_1 segue nel cammino il nodo v_t e quindi deve valere anche che $w(v_t) < w(v_1)$ che è in contraddizione con quanto affermato prima (se un numero è più piccolo di un altro non può contemporaneamente essere più grande dell'altro). Quindi non è possibile che nel cammino monotono ci siano cicli.

ERRORI PIÙ COMUNI:

- Sbagliare l'esempio mettendo i pesi sugli archi invece che sugli nodi: i nodi coinvolti in un arco sono due, se il peso è messo su un arco, a quale nodo si riferisce?
- Sbagliare problema: cercare un cammino monotono considerando i pesi sugli archi
- Cercare il cammino minimo utilizzando Dijkstra/Bellman Ford e pesi sugli archi