Problem Set

Francesco Cosciotti, franci2002.fc@gmail.com, 0323545;
Francesco Mongelli, francesco.mongelli03@libero.it, 0327829;
Alessio Fortini, fortinialessio1@gmail.com, 0324425

December 2023

1 Problema 1

1.1 Risoluzione del problema per $O(n^2)$

Per dimostrare che $O(n^2)$ scambi siano sempre sufficienti, abbiamo pensato a un algoritmo piuttosto banale che, con un costo quadratico, riordini tutte le pinte.

Innanzitutto, abbiamo visto il bancone con le birre come un array $L[1 : n]$ di record t.c. $Gen(L[k]) = \{M, F\}$ e $Beer(L[k]) = \{A, B\}$.

Vedere l'array in questo modo ci ha fatto capire abbastanza velocemente che le operazioni da effettuare per "riordinare" l'array (per riordinare si intende l'accoppiare ogni birra con il proprietario giusto) erano tre:

1. essere in grado di capire se un'accoppiata fosse giusta o sbagliata;
2. ricercare un'eventuale accoppiata giusta;
3. essere in grado attraverso degli scambi contigui di accoppiarla con il proprietario del giusto genere.

1.1.1 Esecuzione dell'algoritmo

Tutte queste operazioni possono essere svolte attraverso due cicli for annidati:

1. Allo scorrere di un indice i (che va da 1 a n) verranno effettuati dei controlli sulle coppie; se l'accoppiata sarà sbagliata, farà partire i cicli successivi, altrimenti, scorrerà come se niente fosse;
2. Questo ciclo viene fatto partire nel momento in cui i incontra un'accoppiata sbagliata. In questo caso, si fa partire un indice j (che va da i a n) che scorre fintanto che non trova una birra che vada bene per la persona che si trova in $L[i]$;
3. Una volta trovata la birra giusta, j comincia ad effettuare degli scambi a ritroso in modo da portare la birra nella giusta posizione.

1.1.2 Pseudocodice

```
function BEER_CHANGE(L[1 : n])
    for i ← 1 to n do:                                              ▷ O(n)
        if A[i] è una coppia sbagliata then:
            j ← i;
            while Beer(L[j]) non è giusta per Gen(L[i]) do:         ▷ O(n)
                incrementa j;
            end while
            while j > i do:                                         ▷ O(n)
                Scambia Beer(L[j]) con Beer(L[j − 1])
                decrementa j;
            end while
        end if
    end for
end function
```

1.1.3 Calcolo della complessità

Possiamo notare che questo algoritmo, nel caso peggiore, esegua un $O(n \times (n+n)) \to O(n \times 2n) \to O(2n^2) \to O(n^2)$.

1.2 Dimostrazione che nel caso peggiore non si possano fare meno di $\Omega(n^2)$

1.2.1 Istanza

Consideriamo un insieme di n persone, divise equamente in $n/2$ maschi (M) e $n/2$ femmine (F). Supponiamo di avere due tipi di birre, A e B, e ogni maschio deve ricevere una birra di tipo A, mentre ogni femmina deve ricevere una birra di tipo B. Le birre sono disposte come segue:

$$\begin{array}{cccccccc} M & M & \dots & M & | & F & F & \dots & F \\ A & A & \dots & A & | & B & B & \dots & B \end{array}$$

1.2.2 Dimostrazione della complessità

Le birre di tipo A sono posizionate a partire dalla $(n/2+1)$-esima posizione. Ogni successiva birra di tipo A si trova a una distanza di $n/2$ posizioni dalla birra da sostituire. Quindi, per ogni maschio, che è ordinato da 1 a $n/2$, sarà necessario effettuare $n/2$ scambi. Poiché il numero totale di maschi è $n/2$, il numero totale di scambi sarà $\Omega(n^2)$.

1.3 Algoritmo con complessità O(n+k)

1.3.1 Proprietà

Nel nostro studio del problema, abbiamo identificato una proprietà chiave legata agli scambi contigui che si è rivelata molto utile per minimizzare il numero di operazioni. Supponiamo di voler scambiare due valori in un array, uno situato all'inizio (chiamato A) e uno alla fine (chiamato B), con n elementi tra di loro (rappresentati come x_n). L'array iniziale è così strutturato:

$$A \quad | \quad x_n \quad | \quad B$$

Abbiamo osservato che spostare A nella posizione di B provoca uno "shifting" delle posizioni a sinistra di tutti gli elementi scambiati con A. Ciò porta all'evoluzione dell'array come segue:

$$x_n \quad | \quad B \quad | \quad A$$

Successivamente, al fine di ripristinare l'ordine originale, è sufficiente far risalire B in prima posizione. Ciò comporta uno "shifting" delle posizioni a destra di tutti gli elementi scambiati con B, portando all'array finale:

$$B \quad | \quad x_n \quad | \quad A$$

In altre parole, l'array risulta identico a quello iniziale, con l'unica differenza che A e B sono scambiati di posizione.

1.3.2 Dimostrazione della proprietà

Per dimostrare questa proprietà possiamo utilizzare un'induzione (da 1 a n) sul numero di elementi compresi fra la prima ed ultima posizione:

- per n = 1 sappiamo che la proprietà è verificata in quanto l'array è identico a quello dell'esempio;
- per n = 2 avremo un array del tipo:

  $$A \quad x_1 \quad x_2 \quad B$$

  portando A in ultima posizione eseguendo gli scambi eseguiremo delle operazioni del tipo:

  $$| \quad A \quad \rightarrow \quad x_1 \quad | \quad x_2 \quad B$$

  $$x_1 \quad | \quad A \quad \rightarrow \quad x_2 \quad | \quad B$$

  $$x_1 \quad x_2 \quad | \quad A \quad \rightarrow \quad B \quad |$$

  $$x_1 \quad x_2 \quad B \quad A$$

  una volta in questa posizione facciamo risalire B in questo modo:

  $$x_1 \quad | \quad x_2 \quad \leftarrow \quad B \quad | \quad A$$

  $$| \quad x_1 \quad \leftarrow \quad B \quad | \quad x_2 \quad A$$

  $$B \quad x_1 \quad x_2 \quad A$$

- Assumendo sia vero per ogni n dimostriamolo per $n + 1$

- Essendo n = n+1 l'array avrà una forma del tipo:

$$A \quad x_1 \quad \dots \quad x_n \quad x_{n+1} \quad B$$

contando $x_1 \dots x_2$ come un unico elemento eseguiremo delle operazioni sull'array del tipo:

$$| \quad A \quad \rightarrow \quad x_1 \quad \dots \quad x_n \quad | \quad x_{n+1} \quad B$$

$$x_1 \quad \dots \quad x_n \quad | \quad A \quad \rightarrow \quad x_{n+1} \quad | \quad B$$

$$x_1 \quad \dots \quad x_n \quad x_{n+1} \quad | \quad A \quad \rightarrow \quad B \quad |$$

$$x_1 \quad \dots \quad x_n \quad x_{n+1} \quad B \quad A$$

arrivati ad avere questa posizione eseguiremo le operazioni necessarie a far risalire B:

$$x_1 \quad \dots \quad x_n \quad | \quad x_{n+1} \quad \leftarrow \quad B \quad | \quad A$$

$$| \quad x_1 \quad \dots \quad x_n \quad \leftarrow \quad B \quad | \quad x_{n+1} \quad A$$

$$B \quad x_1 \quad \dots \quad x_n \quad x_{n+1} \quad A$$

avendo ora un array di questo tipo possiamo avvermare di poter ritenere vera questa proprietà avendola verificata per $\forall n > 0$;

1.3.3 Idea alla base di un algoritmo attorno a questa proprietà

Dopo aver dimostrato la proprietà che semplifica gli scambi tra coppie di elementi adiacenti all'interno di un array, abbiamo deciso di applicarla al nostro problema specifico: ordinare le coppie sbagliate di genere diverso, come {M, A} con {F, B}. Questa strategia ci consente di ordinare due pinte dopo solo due passaggi attraverso l'array, portando a una complessità di tipo $O(k \times n)$, dove k è il numero di coppie da correggere.

Successivamente, abbiamo creato un array ausiliario $P[1 : k]$, dove k è il numero di coppie da correggere. In questo array, abbiamo inserito le coppie maschio-femmina con pinte sbagliate, organizzandole in modo che i maschi occupino posizioni con indici dispari e le femmine occupino posizioni con indici pari. Questo processo di creazione di P richiede un tempo $O(n)$.

Una volta creato P, abbiamo riassegnato le posizioni delle coppie in modo che l'indice minore della coppia fosse in una posizione dispari e l'indice maggiore in una posizione pari.

Infine, abbiamo ciclato attraverso la lunghezza di P (operazione $O(k)$) per determinare quali coppie scambiare, sfruttando la proprietà dimostrata in precedenza.

Questo approccio ci consente di correggere le coppie sbagliate in modo efficiente e sistemico.

1.3.4 Creazione dell'algoritmo

Avendo fisate tutte queste idee ora bisogna soltanto applicarle nel modo giusto infatti:

1. Scorriamo l'array L al fine di individuare le birre mal assegnate, salvandole nell'array ausiliario P e mantenendo invariate le proprietà degli indici. (Complessità: $O(n)$)

2. Successivamente, scorriamo l'array P riordinando le posizioni delle coppie in modo che il valore minore preceda sempre quello maggiore. (Complessità: $O(k)$)

3. Ora scorriamo P a coppie, salvando gli indici delle coppie in posizioni dispari e pari rispettivamente. (Complessità: $O(k)$)

4. Dopo aver memorizzato gli indici, applichiamo la proprietà dimostrata in precedenza utilizzando due cicli per effettuare gli scambi necessari. (Complessità: $O(n+n)$)

1.3.5 pseudocodice

function BeerChange(L[1:n]) Inizializza P[1:n] con tutti zero
  $t \leftarrow 1$
  $m \leftarrow 1$
  $f \leftarrow 2$
  for $t < n$ **do**: ▷ O(n)
    if se Gen(L[t]) è M e ha una birra sbagliata **then**:
      $P[m] \leftarrow t$
      Incremento m di due; ▷ m è sempre dispari
    else if se Gen(L[t]) è F e ha una birra sbagliata **then**:
      $P[f] \leftarrow t$
      Incremento f di due; ▷ f è sempre pari
    end if
  end for
  $w \leftarrow 1$
  while $P[w] \neq 0$ **do** ▷ O(k)
    if $P[w+1] < P[w]$ **then**
      Scambia;
    end if
    Incremento k di due ▷ vado a vedere solo le coppie
  end while
  $j \leftarrow 1$
  while $P[j] \neq 0$ **do**: ▷ O($\frac{k}{2}$)
    $u \leftarrow P[j]$
    $v \leftarrow P[j+1] - 1$
    while $u < P[j]$ **do**: ▷ O(n)
      Scambia Beer(L[u]) con Beer(L[u+1]);
      Incremento u;
    end while
    while $v > P[j]$ **do**: ▷ O(n)
      Scambia Beer(L[v]) con Beer(L[v-1]);
      Decremento v;
    end while
    Incremento j di due;
  end while
end function

1.3.6 Calcolo della coplessità

Nel caso medio, quando $k < n$, l'algoritmo ha una complessità di tipo $O(n + k)$. Ciò è dovuto al fatto che la creazione di P ha un costo di $O(n)$, il riordinamento di P ha un costo di $O(k)$, e il terzo ciclo ha una complessità di $O\left(\frac{k}{2} \times 2n\right)$, ovvero $O(k \times n)$. Pertanto, la complessità totale dell'algoritmo è $O(n + k + k \times n)$, che può essere semplificata a $O(n + k \times n)$.

Nel caso peggiore, quando $k = n$, la complessità dell'algoritmo diventa $O(n^2)$, poiché il terzo ciclo avrà una complessità di $O(n^2)$ al posto di $O(k \times n)$.

In sintesi, l'algoritmo presenta una complessità che dipende dalla relazione tra n e k, e nel caso medio risulta più efficiente rispetto al caso peggiore.

2 Problema 2

La prima intuizione è stata di rappresentare la griglia $n \times m$ come un array $A[1 : n]$, dove ogni posizione $A[i]$ ($\forall i \in [1, n]$) indica l'altezza della colonna. Quando si posiziona un blocco, è necessario scorrere due volte l'array da x a z (dove z è l'angolo destro del blocco, calcolato come $x + w$): una per trovare l'altezza massima, chiamata k, e una seconda per sostituire i valori con $k + h$. Successivamente, si controlla se $k + h$ supera m. In tal caso, l'algoritmo restituisce l'indice della mossa in cui è stato posizionato l'ultimo blocco; altrimenti, continua a scorrere le mosse.

È evidente che questo approccio ha una complessità asintotica di $O(Nn)$ a causa della crescita lineare della lunghezza del blocco, mentre l'obiettivo è raggiungere una complessità di $O(N\sqrt{n})$.

2.1 Idea alla base dell'algoritmo finale

Con il suggerimento del professore Luciano Gualà, abbiamo capito che per ottenere una lunghezza lineare di $\sqrt{n}$, dovevamo suddividere l'array A in $\sqrt{n}$ sezioni, chiamate "chunk", ciascuna di lunghezza $\sqrt{n}$. Questa partizione ci consente di salvare l'altezza massima di ogni chunk in un array $B[1 : \sqrt{n}]$, dove ogni posizione $B[k]$ ($\forall k \in [1, \sqrt{n}]$) rappresenta l'altezza massima all'interno del k-esimo chunk.

Utilizzando l'array B, possiamo notare che la ricerca del massimo può essere eseguita in modo più efficiente, dividendo il blocco in tre parti:

1. La prima sezione va dall'indice x fino alla fine del chunk in cui si trova (chiamato α), calcolato come $\left\lfloor \frac{x}{\sqrt{n}} \right\rfloor$;
2. La seconda va dall'inizio del chunk successivo a quello di x fino al chunk immediatamente prima di quello in cui si trova z;
3. La terza va dall'inizio del chunk in cui si trova z (chiamato ω), calcolato come $\left\lfloor \frac{z}{\sqrt{n}} \right\rfloor$, fino alla posizione effettiva di z.

Questa divisione consente di cercare il massimo in $A[x : (\alpha+1)\sqrt{n}-1]$, in $B[\alpha+1 : \omega-1]$, e infine in $A[\omega\sqrt{n} : z]$. La ricerca del massimo k può quindi essere eseguita in $O(\sqrt{n})$, poiché le lunghezze delle prime e terze sezioni sono al massimo $\sqrt{n}$, così come lo è B. Successivamente, sostituiremo i valori in $A[x : (\alpha + 1)\sqrt{n} - 1]$, $B[\alpha + 1 : \omega - 1]$, e $A[\omega\sqrt{n} : z]$, assicurandoci di sostituire i valori in B se $k + h$ supera i valori attualmente presenti in B. Grazie a questi scorrimenti, garantiamo che le liste A e B siano sempre aggiornate al momento in cui viene posizionato un blocco, portando la complessità dell'algoritmo a $O(N\sqrt{n})$.

2.2 Specifiche riguardo l'algoritmo finale

L'algoritmo progettato potrebbe essere sufficiente, ma abbiamo voluto aggiungere ulteriori casistiche e un array ausiliario aggiuntivo per velocizzare ulteriormente la ricerca del massimo e la sua sostituzione.

Innanzitutto, abbiamo introdotto un secondo array ausiliario chiamato $C[1 : \sqrt{n}]$, in cui ogni posizione $C[j]$ ($\forall j \in [1 : \sqrt{n}]$) indica tramite un valore booleano se il j-esimo chunk è spianato o meno (True se l'altezza di tutte le colonne appartenenti al chunk è uguale, False altrimenti).

Le diverse casistiche sono le seguenti:

- Il blocco occupa un intero chunk: la ricerca del massimo è istantanea grazie a B, e i valori vengono sostituiti interamente in A attraverso un ciclo, con un costo di $O(\sqrt{n})$. Successivamente, il valore di C relativo al chunk viene impostato su True.
- Il blocco cade in un unico chunk ma non lo occupa interamente: si cerca il massimo in $A[x : z]$ (essendo $z - x < \sqrt{n}$ per costruzione), e poi si sostituiscono i valori ciclando in questo intervallo. Bisogna fare attenzione a sostituire il valore relativo al chunk in B e a negare quello in C se necessario.
- Il blocco occupa più di un chunk: questo caso è gestito come spiegato nella sezione precedente, con l'aggiunta che bisogna impostare il valore relativo ai chunk intermedi in C e negare quelli relativi al chunk iniziale e finale.

2.3 Pseudocodice

function GameOverChecker(N, n, m)
  Inizializza $B[1 : \sqrt{n}]$ con tutti zero
  Inizializza $C[1 : \sqrt{n}]$ con tutti True
  $i \leftarrow 1$ ▷ Variabile per il conteggio delle mosse
  for i to N **do** ▷ O(N)
    $x \leftarrow x_i$
    $z \leftarrow x_i + w_i$
    $Chunk_i \leftarrow \lfloor \frac{x}{\sqrt{n}} \rfloor$
    $Chunk_e \leftarrow \lfloor \frac{z}{\sqrt{n}} \rfloor$
    $k \leftarrow 0$
    if $Chunk_i = Chunk_e$ and $w_i = \sqrt{n}$ **then**
      $k \leftarrow B[Chunk_i] + h_i$
      for $t \leftarrow x$ to z **do** ▷ O($\sqrt{n}$)
        $A[t] \leftarrow k$
      end for
      $C[Chunk_i] \leftarrow True$
    end if
    if $Chunk_i = Chunk_e$ **then** ▷ da vedere come se fosse un else if
      $k \leftarrow \max(A[x : z]) + h_i$ ▷ O($\sqrt{n}$)
      for $j \leftarrow x$ to z **do** ▷ O($\sqrt{n}$)
        $A[j] \leftarrow k$
      end for
      if $B[Chunk_i] < k$ **then**

```
            B[Chunk_i] ← k
        end if
        C[Chunk_i] ← False
    else
        if C[Chunk_i] = True then
            k ← B[Chunk_i]
        else
            k ← max(A[x : ((Chunk_i + 1) × √n) − 1])                    ▷ O(√n)
        end if
        if k < max(B[Chunk_i + 1 : Chunk_e − 1]) then
            k ← max(B[Chunk_i + 1 : Chunk_e − 1])                       ▷ O(√n)
        end if
        if C[Chunk_e] = True and k < B[Chunk_e] then
            k ← B[Chunk_e]
        else
            k ← max(A[Chunk_e × √n : z])                                ▷ O(√n)
        end if
        k ← k + h_i
        α ← x
        for α to ((Chunk_i + 1) × √n) − 1  do                           ▷ O(√n)
            A[α] ← k
        end for
        if k > B[Chunk_i] then
            B[Chunk_i] ← k
        end if
        β ← Chunk_i + 1
        for β to Chunk_e − 1 do                                         ▷ O(√n)
            B[β] ← k
        end for
        γ ← Chunk_e × √n
        for γ to z do                                                   ▷ O(√n)
            A[γ] ← k
        end for
        if k > B[Chunk_e] then
            B[Chunk_e] ← k
        end if
        C[Chunk_i] ← False
        C[Chunk_e] ← False
    end if
    if k > m then return l
    end if
  end for
return Il gioco continua
end function
```

3 Problema 3

3.1 Distance

3.1.1 Costruzione struttura dati

Affinché la nostra query possa mantenere un costo temporale O(1), dovremo avere accesso preventivo a qual è la lunghezza di ciascun nodo. Ciò, in realtà, è abbastanza semplice: ci basterà costruire, dato un albero T di n nodi, un array indicizzato di n celle, dove ogni cella contiene:

- Un nodo.
- Il peso associato al nodo, inteso come la somma degli archi sul cammino, d'ora in poi indicato come weight(v) dato un generico nodo v.

Agli scopi di questa spiegazione, l'ordinamento dell'array è irrilevante: presupporremo di avere già un riferimento esplicito alla loro posizione.

3.1.2 Query

Una volta palesata la struttura dati, è relativamente semplice intuire come calcolare in tempo 0(1) la lunghezza tra due nodi. Risulterà, infatti:

function DIST(u, v)
  return $weight(v) - weight(u)$
end function

3.2 Level Ancestor

3.2.1 Considerazioni iniziali e algoritmo banale

Posto inizialmente il problema di rintracciare un antenato/figlio in base ad un qualsivoglia criterio, la risposta banale è ovvia: scorriamo la nostra struttura dati fino a che non troviamo il nodo bersaglio. Questa soluzione, sebbene corretta, non rispetta la necessità di creare una query che restituisca in tempo $O(\log(n))$ la risposta alla nostra domanda, attenendosi, nel caso peggiore, ad un O(n). Per ottenere, dunque, il tempo desiderato, dovremo avere dei riferimenti espliciti già in fase di creazione della struttura dati.

3.2.2 Costruzione struttura dati

Per guadagnare una maggiore efficienza in fase di ricerca, dato un qualsiasi albero T di n nodi, struttureremo la base di dati in due parti fondamentali.

- Un array indicizzato di lunghezza n, t.c. ogni cella contenga un nodo dell'albero. L'ordinamento, per gli scopi di questa spiegazione, è irrilevante: si presupporrà che, al momento della query, si sappia già dove è locato il nodo di partenza v.
- Per ogni cella dell'array, successivamente, verrà definita un "vettore degli antenati", ossia un vettore di puntatori, t.c. ciascuno indichi, se esiste, l'antenato lontano 2^i archi dal nodo.

Avremo dunque, per un qualsiasi nodo u, un riferimento esplicito all'antenato in posizione 0 (se stesso), 1, 2, 4, 8, ecc.

Per calcolare la dimensione totale della struttura, ci basta osservare che ogni "vettore degli antenati" consisterà di i elementi, per il massimo i t.c. 2^i sia al più n. Poiché, salvando solo gli antenati lontani potenze di 2 archi, e considerando la rappresentazione binaria di n, tali antenati saranno al più il numero di cifre necessarie a rappresentare n in binario (visto che salvo solo gli antenati del tipo 1_2, 10_2, 100_2, ecc.), quindi ogni vettore avrà dimensione $O(\log(n))$. Ripetendo il processo per ogni nodo nell'array di lunghezza n, la dimensione finale sarà $O(n \log(n))$.

3.2.3 Query

La fase di query sfrutterà la strutturazione della base dati per ritornare in tempo $O(\log(n))$ il nostro nodo bersaglio. LA(v,k) sarà una chiamata ricorsiva, che consterà di 3 esiti possibili:

- L'antenato a livello k non esiste. Caso di fallimento: restituiremo semplicemente NULL. Facilmente strutturabile con un qualche tipo di check all'inizio del codice.
- L'antenato a livello k esiste, e k è della forma 2^i per un qualche intero i. Caso base: ritorneremo il nodo associato al puntatore corrispondente nel vettore degli antenati.
- L'antenato a livello k esiste, ma k non è della forma 2^i per un qualche intero i. Caso ricorsivo: ritorneremo una nuova chiamata LA(v, k_i), con $k_i = k - q$ con q la più grande potenza di 2 minore di k.

3.2.4 Pseudocodice

LA(k,v)
Sia T un albero di n nodi
if v è la radice di T **and** $k > 0$ **then** //Se non esiste
return NULL
else
$q = \lfloor \log_2 k \rfloor$
$u \leftarrow$ nodo del vettore degli antenati in posizione q
if $q = k$ **then**
return u
else
return LA$(u, k - q)$
end if
end if

3.2.5 Costo

Affinché si possa analizzare il costo dell'algoritmo di query, è bene scomporne nuovamente il funzionamento in base ai possibili esiti:

- Nel caso in cui l'antenato esista e risulti $k = 2^i$ per un qualche intero i, il costo temporale dell'algoritmo sarà $O(1)$.

- Nel caso in cui l'antenato esista e risulti che k non sia potenza di 2, si avvierà una serie di chiamate ricorsive, di cui possiamo studiare il costo temporale facendo queste considerazioni: ogni chiamata ha costo $O(1)$; ogni numero binario può essere scomposto in somme di numeri binari che rappresentano potenze di 2 (ad esempio: 1001101_2 può essere scomposo in $1000000_2 + 1000_2 + 100_2 + 1_2$). Quindi per ogni k che non è potenza di 2 la funzione verrà richiamata al più per ogni "1" presente nella rappresentazione binaria di k, che è al più il numero di cifre necessario per rappresentare k in binario, ovvero $O(\log(k))$, ma, visto che in questo caso $k = O(n)$ (poiché l'antenato esiste), il costo temporale sarà $O(\log(n))$.

3.3 Weighted Level Ancestor

3.3.1 Costruzione struttura dati

Per questo punto del terzo problema abbiamo pensato, sostanzialmente, ad una combinazione dei precedenti due punti. Infatti, per ogni nodo v dell'albero T salveremo:

- Il suo peso dalla radice (inteso come dist(r, v), con r radice di T) che quindi chiameremo weight(v), come informazione satellite all'interno del nodo stesso.
- Un vettore indicizzato di antenati (che chiamo $v.antenati$), che segue lo stesso criterio del secondo punto del problema (contiene solo gli antenati distanti potenze di 2).

In questo modo la struttura dati avrà complessita spaziale $O(nlogn)$ poiché (come nei punti precedenti) per ogni nodo salvo una informazione satellite e un vettore di antenati di grandezza $O(logn)$ (dimostrato nel punto precedente). Agli scopi di questa spiegazione, l'ordinamento dell'array è irrilevante: presupporremo di avere già un riferimento esplicito alla loro posizione.

3.3.2 Query

La fase di risposta alla query segue la falsa riga del secondo punto: $WLA(v, \Delta)$ sarà una chiamata ricorsiva, che:

- Elaborerà l'antenato u meno profondo di v nel suo vettore degli antenati tale che $dist(u, v) < \Delta$.
- Richiamerà la funzione $WLA(u, \Lambda)$, con u come definito prima e $\Lambda = \Delta - dist(u, v)$.

Il caso base da verificare prima di richiamare la funzione si verifica se il nodo u è il nodo v, in tal caso la funzione ritornerà il padre di v. Poiché so che v è il suo antenato meno profondo tale che $dist(u, v) < \Delta$, e al contempo ho verificato se suo padre rispetta la proprietà richiesta, essendo un suo antenato distante $1(= 2^0)$ e ho scoperto che $dist(padre(v), v) \geq \Delta$.

3.3.3 Pseudocodice

$WLA(v, \Delta)$
Sia T un albero di n nodi
if v è la radice di T **and** $\Delta > 0$ **then** //Se non esiste
return NULL
end if

$i = 0$
while $dist(v.antenati[i], v) < \Delta$ **do**
    $i \mathrel{+}= 1$
end while
$u \leftarrow v.antenati[i]$
if $v = u$ **then**
    return $padre(v)$
else
    $\Lambda = \Delta - dist(u, v)$
    return $WLA(u, \Lambda)$
end if

3.3.4 Costo

Per studiare la complessità temporale dell'algoritmo basta fare un parallelismo con i punti precedenti:

- Ogni operazione di lettura del vettore degli antenati e ogni chiamata della funzione $dist(u, v)$ (la funzione creata nel primo punto) costa $O(1)$, infine il ciclo *while* gira al più $O(\log n)$.
- Per motivi paralleli a quelli del secondo punto al massimo la funzione verrà richiamata $O(\log n)$ volte.

Quindi, in totale, l'algoritmo avrà complessita temporale $O(\log n \log n)$***, che non è la specifica richiesta dal problema, non siamo riusciti a risolvere il terzo punto causa tempistiche e impegni, ma ci abbiamo provato e abbiamo comunque voluto presentare la miglior risoluzione trovata dal gruppo, anche se sbagliata o incompleta.***