Problem Set 3

Cosciotti Francesco (0323545) Porzia Adriano (0327131) Zbirciog Ionut Georgian (0308984)

Maggio 2024

Indice

1 Problema 2 *(Un gioco con pedine e scacchiera)* **2**
1.1 Definizione dei sottoproblemi . . . 2
1.2 Casistiche . . . 2
1.3 Formula di Bellman . . . 4
1.4 Pseudocodice . . . 5

2 Problema 3 *(Costruire un albero di costo minimo)* **6**
2.1 Definizione dei sottoproblemi . . . 6
2.2 Struttura dati . . . 6
2.3 Combinazione dei sottoproblemi . . . 6
2.4 Formula di Bellman . . . 7
2.5 Pseudocodice . . . 8
2.6 Ricostruzione dell'albero . . . 9

1 Problema 2 *(Un gioco con pedine e scacchiera)*

1.1 Definizione dei sottoproblemi

Per la risoluzione del problema, definiamo i sottoproblemi come segue: per ogni posizione $(i,\ j)$ all'interno di una scacchiera, considerata come una matrice di dimensioni $n \times 3$, cerchiamo di determinare lo score massimo che possiamo ottenere posizionando una pedina in quella specifica casella.

In altre parole, per ogni cella della scacchiera rappresentata dalle coordinate $(i,\ j)$, vogliamo calcolare il punteggio massimo che possiamo ottenere posizionando una pedina in quella cella, tenendo conto delle possibili pedine già posizionate in altre celle della scacchiera. Questo processo di calcolo del punteggio massimo per ogni cella costituisce il nucleo dei sottoproblemi che affronteremo nell'algoritmo di programmazione dinamica.

$OPT[i,\ j]$ **è il massimo score ottenibile posizionando la pedina nella casella** $(i,\ j)$.

Struttura dati

Inizialmente abbiamo concepito OPT come una matrice di dimensioni $n \times 3$, dove ogni posizione rappresentava una cella della scacchiera che contenesse il massimo punteggio ottenibile posizionando una pedina in quella cella. Tuttavia, combinando i vari sottoproblemi, abbiamo riconosciuto la necessità di utilizzare 2 vettori aggiuntivi insieme alla matrice OPT.

- $OPT_{COMB}[1 : n]$: Questo array contiene i valori *combinati* (vedere paragrafo successivo) delle colonne della matrice OPT. Ogni elemento $OPT_{COMB}[j]$ rappresenta il risultato ottenuto da una specifica casistica della formula di Bellman, applicata alla $j - esima$ colonna della matrice OPT.
- $OPT_{MAX}[1 : n]$: Questo array memorizza il massimo tra i tre valori della $j - esima$ colonna di OPT e il valore corrispondente di OPT_{COMB}. In altre parole, per ogni indice j, $OPT_{MAX}[j]$ rappresenta il massimo tra i punteggi della $j - esima$ colonna di OPT e il valore combinato della stessa colonna rappresentato da $OPT_{COMB}[j]$.

In sintesi, questi due vettori aggiuntivi, OPT_{COMB} e OPT_{MAX}, sono essenziali per gestire in modo efficiente i punteggi dei sottoproblemi e per calcolare il punteggio massimo complessivo in ciascun passaggio dell'algoritmo.

1.2 Casistiche

Dopo aver definito la struttura dati e il contenuto di OPT, rimane solo di comprendere come combinare efficacemente i sottoproblemi. Per evitare ridondanze nei dati raccolti in OPT, possiamo generalizzare i vari casi in una matrice 3×3, visto che collezionando lo score delle pedine più lontane, rischieremmo di avere valori duplicati poiché sono già inclusi negli OPT più vicini.

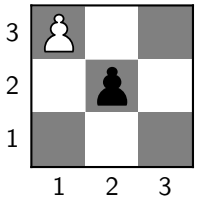
Per capire come combinare i sottoproblemi abbiamo analizzato quello che è il principale limite sul posizionamento delle pedine, infatti inizialmente abbiamo pensato alle seguenti combinazioni.

Sia i l'indice della $i - esima$ riga e sia j l'indice della $j - esima$ colonna.
Siano le pedine nere i valori di OPT già calcolati e sia la pedina bianca il valore da aggiungere ad OPT da calcolare.

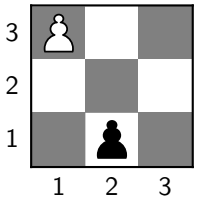
Casi banali

Per analizzare i casi banali, supponiamo che le soluzioni delle colonne 2 e 3 sono state già calcolate, ovvero sono presenti nella matrice OPT e nei vettori OPT_{MAX} e OPT_{COMB}.

- Volendo collezionare lo score della casella $(i, j) = (3, 1)$ in cui è posizionata la pedina bianca, per come è definito il problema, nella collona a destra possiamo collezionare il valore delle celle in cui sono posizionate le pedine nere nelle seguenti posizioni:

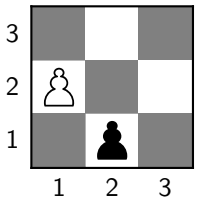


$OPT[3,1] = OPT[2,2] + score[3,1]$

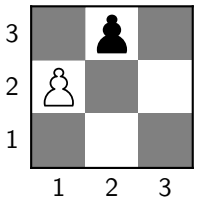


$OPT[3,1] = OPT[1,2] + score[3,1]$

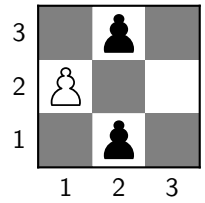
- Volendo collezionare lo score della casella $(i, j) = (2, 1)$ in cui è posizionata la pedina bianca, per come è definito il problema, nella collona a destra possiamo collezionare il valore delle celle in cui sono posizionate le pedine nere nelle seguenti posizioni:



$OPT[2,1] = OPT[1,2] + score[2,1]$

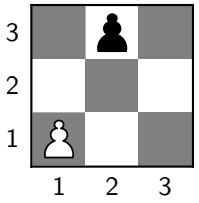


$OPT[2,1] = OPT[3,2] + score[2,1]$

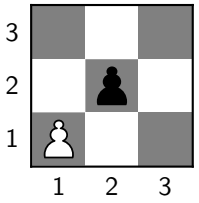


$OPT[2,1] = OPT[1,2] + OPT_{COMB}[2] + score[2,1]$

- Volendo collezionare lo score della casella $(i, j) = (1, 1)$ in cui è posizionata la pedina bianca, per come è definito il problema, nella collona a destra possiamo collezionare il valore delle celle in cui sono posizionate le pedine nere nelle seguenti posizioni:



$OPT[1,1] = OPT[3,2] + score[1,1]$

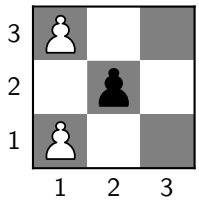


$OPT[1,1] = OPT[2,2] + score[1,1]$

Caso combinato

Successivamente, abbiamo notato l'esistenza di un'ulteriore caso, che chiameremo *caso combinato*, ovvero il caso in cui andiamo a prendere sia il valore nella riga $i = 1$ che quello nella riga $i = 3$, il cui risultato viene poi salvato all'interno dell'array $OPT_{COMB}[j]$ dove j indica la colonna.

In questo caso vogliamo andare a collezionare lo score delle pedine in posizione $(i, j) = (3, 1)$ e $(1, 1)$.

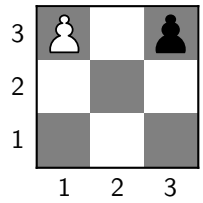


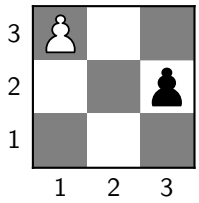
$OPT_{COMB}[1] = OPT[2,2] + score[3,1] + score[1,1]$

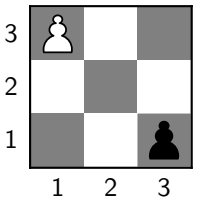
Caso in cui escludiamo una colonna

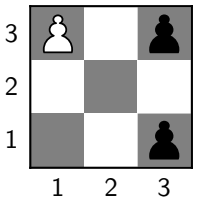
Inoltre, analizzando l'istanza data, abbiamo notato la possibilità di dover saltare una colonna k, poiché il massimo score possibile può essere ottenuto solo combinando valori incompatibili presenti nelle colonne $k + 1$ e $k - 1$.

Per ogni cella $(i, 1)$ con $i \in \{1, 2, 3\}$ abbiamo le seguenti casistiche:

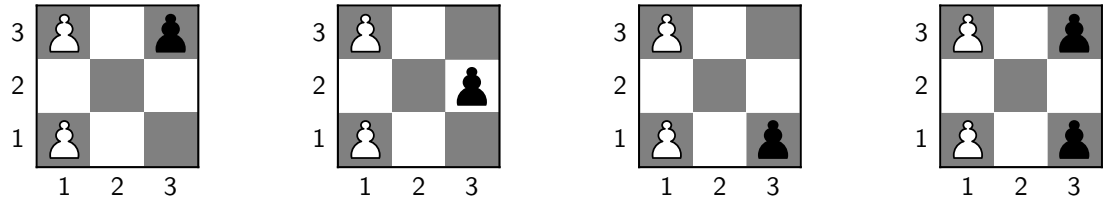








Inoltre bisogna tenere conto anche del caso in cui andiamo a combinare le celle, dunque otteniamo:



Tuttavia, se manteniamo il massimo per ogni colonna in OPT_{MAX}, questi casi possono essere semplificati in uno solo, specificato poi nella formula di Bellman.

Si osserva facilmente che escludere un numero maggiore di colonne non produce alcun guadagno.

Riepilogo

Riassumendo, le nostre possibili casistiche:

- Per $j = 1$ abbiamo 2 casi banali e il caso in cui si esclude una colonna.
- Per $j = 2$ abbiamo 3 casi banali e il caso in cui si esclude una colonna.
- Per $j = 3$ abbiamo 2 casi banali e il caso in cui si esclude una colonna.
- Per il combinato, abbiamo un caso banale e il caso in cui si esclude una colonna.

Numero dei sottoproblemi

Osservando il numero di sottoproblemi per ogni colonna, notiamo che questo è uguale a 4. Moltiplicando questo numero per il numero totale di colonne, otteniamo un totale di sottoproblemi pari a $4n = O(n)$ nella dimensione dell'input.

Goal

Poiché le celle della prima colonna sono le ultime da analizzare all'interno della matrice, il punteggio massimo trovato in questa colonna corrisponde al punteggio massimo per l'intera scacchiera. Questo valore, infatti, è quello salvato all'interno di $OPT_{MAX}[1]$.

1.3 Formula di Bellman

Caso base

Il nostro caso base è la colonna n dove non abbiamo bisogno di combinare gli score con altri valori di OPT.

$$\begin{cases} OPT[i,n] = score[i,n] \qquad \forall\ i = 1,2,3 \\ OPT_{COMB}[n] = score[1,n] + score[3,n] \\ OPT_{MAX}[n] = \max\{OPT[1,n],\ OPT[2,n],\ OPT[3,n],\ OPT_{COMB}[n]\} \end{cases}$$

Casi generici

$$\begin{cases} OPT[1,i] = \max\{OPT[2,i+1],\ OPT[3,i+1],\ OPT_{MAX}[i+2]\} + score[1,i] \\ OPT[2,i] = \max\{OPT[1,i+1],\ OPT[3,i+1],\ OPT_{COMB}[i+1],\ OPT_{MAX}[i+2]\} + score[2,i] \\ OPT[3,i] = \max\{OPT[1,i+1],\ OPT[2,i+1],\ OPT_{MAX}[i+2]\} + score[3,i] \\ OPT_{COMB}[i] = \max\{OPT[2,i+1],\ OPT_{MAX}[i+2]\} + score[1,i] + score[3,i] \\ OPT_{MAX}[i] = \max\{OPT[1,i],\ OPT[2,i],\ OPT[3,i],\ OPT_{COMB}[i]\} \end{cases}$$

1.4 Pseudocodice

Algorithm 1 Algoritmo_Scacchiera

```
1:  function COMPUTE_OPT(score)
2:                                                  ▷ n + 1 per non sforare nella formula
3:      Matrix OPT[3, (n + 1)]
4:      Array OPT_MAX[n + 1]
5:      Array OPT_COMB[n + 1]
6:                                                  ▷ Caso base
7:      for i = 1 to 3 do
8:          OPT[i, n] = score[i, n]
9:      end for
10:     OPT_COMB[n] = score[1, n] + score[3, n]
11:     OPT_MAX[n] = max{OPT[1, n], OPT[2, n], OPT[3, n], OPT_COMB[n]}
12:                                                 ▷ Fine caso base
13:                                                 ▷ Casi Generici
14:     for i = n − 1 down to 1 do
15:         OPT[1, i] = max{OPT[2, i + 1], OPT[3, i + 1], OPT_MAX[i + 2]} + score[1, i]
16:
17:         OPT[2, i] = max{OPT[1, i + 1], OPT[3, i + 1], OPT_COMB[i + 1], OPT_MAX[i + 2]} + score[2, i]
18:
19:         OPT[3, i] = max{OPT[1, i + 1], OPT[2, i + 1], OPT_MAX[i + 2]} + score[3, i]
20:
21:         OPT_COMB[i] = max{OPT[2, i + 1], OPT_MAX[i + 2]} + score[1, i] + score[3, i]
22:
23:         OPT_MAX[i] = max{OPT[1, i], OPT[2, i], OPT[3, i], OPT_COMB[i]}
24:     end for
        return OPT_MAX[1]
25: end function
```

Analisi della complessità

- Il tempo di esecuzione dell'algoritmo è una diretta conseguenza del numero dei sottoproblemi. Infatti avendo $\theta(n)$ sottoproblemi, ciascuno risolvibile in tempo $O(1)$, segue che l'algoritmo trova la soluzione ottima in tempo $\theta(n)$, dove n è il numero di colonne della scacchiera.
- Per quanto riguarda lo spazio utilizzato, usando la tecnica tabulation, in memoria salviamo una matrice $3 \times n$ più 2 array ausiliari entrambi di lunghezza n. Di conseguenza, l'algoritmo calcola la soluzione ottima in spazio $\theta(n)$, dove n è il numero di colonne della scacchiera.

2 Problema 3 *(Costruire un albero di costo minimo)*

2.1 Definizione dei sottoproblemi

Nella risoluzione del problema, abbiamo focalizzato il nostro ragionamento sulle proprietà fondamentale che deve avere un albero per essere accettabile, ovvero:

1. Il costo complessivo dell'albero deve essere minimo.
2. Ogni nodo interno deve avere esattamente due figli.
3. L'albero ha esattamente n foglie.

$OPT[i,j]$ rappresenta l'albero di costo minimo contenente le foglie $\{C[j], C[j+1], ..., C[j+i-1]\}$ date in input, $\forall i \in \{1, ..., n\} \wedge \forall j \in \{1, ..., n-i+1\}$ dove i indica la $i-esima$ riga e j indica la $j-esima$ colonna della matrice OPT.

2.2 Struttura dati

Abbiamo definito una matrice OPT di dimensioni $n \times n$, dove ogni cella contiene 4 valori:

- $OPT[i,j].val$: il costo minimo per costruire un albero con le foglie che vanno da j a $j+i-1$ (dove i rappresenta le righe e j le colonne).
- $OPT[i,j].max$: il valore massimo tra le foglie all'interno dell'intervallo $[j : j+i-1]$.
- $OPT[i,j].sin$: le coppie di indici $i,\ j$ che all'interno di OPT ci indicano quale cella guardare per ricostruire la struttura del sottoalbero sinistro.
- $OPT[i,j].des$: le coppie di indici $i,\ j$ che all'interno di OPT ci indicano quale cella guardare per ricostruire la struttura del sottoalbero destro.

Tuttavia, è importante notare che utilizziamo solo la metà delle celle di OPT. In particolare, ci limitiamo alla matrice triangolare superiore sinistra, poiché oltre questa zona andremmo oltre il numero di foglie salvate nell'array C, contenente i valori delle foglie dati in input.

2.3 Combinazione dei sottoproblemi

Nel processo di combinazione dei sottoproblemi, abbiamo tenuto presente che un albero ottimale deve essere strutturato in modo binario, cioè deve avere due sottoalberi per ogni nodo interno. Questa considerazione ci ha portato a distinguere le foglie in due gruppi: le "foglie destre", che appartengono al sottoalbero destro, e le "foglie sinistre", che appartengono al sottoalbero sinistro.

Casi base

Durante l'analisi dei casi base, abbiamo individuato due situazioni:

- Il primo caso base riguarda la prima riga della matrice, dove ogni cella rappresenta una foglia singola. In questo caso, assegniamo il valore della foglia a $OPT[1,j].max$, mentre poniamo il valore di $OPT[1,j].val = 0$ (poiché per la nostra definizione le foglie non hanno valore). $OPT[1,j].sin$ e $OPT[1,j].des$ non vengono inizializzate in quanto non ci sono sottoalberi.
- Il secondo caso base si riferisce alla seconda riga della matrice, dove ogni cella rappresenta un albero con solo due foglie. Questo albero può essere combinato in un unico modo, poiché le foglie devono essere ordinate secondo una visita simmetrica rispetto all'input. In questo caso, il valore di $OPT[2,j].val$ è il prodotto tra gli $OPT[1,j].max$ e $OPT[1,j+1].max$ i quali indicano le foglie, mentre $OPT[2,j].max$ rappresenta il massimo tra i due valori delle foglie. Inoltre, assegniamo gli indici delle foglie a $OPT[2,j].sin$ e $OPT[2,j].des$ per poter trovare facilmente le loro posizioni nella matrice.

Caso generico

Avendo effettuato questa suddivisione tra sottoalbero destro e sottoalbero sinistro, ci siamo resi conto che la chiave per trovare il valore dell'albero ottimale consiste nel provare tutte le possibili suddivisioni fra foglie destre e sinistre e trovare quella con il costo complessivo minimo. Questo si riduce a cercare un indice k nell'intervallo $[1 : i-1]$ tale che il nodo risultante dalla combinazione del sottoalbero sinistro contenente le foglie da j a $j+k$ e del sottoalbero destro contenente le foglie da $j+k+1$ a $j+i-1$ sia il più piccolo possibile in termini di costo.

Numero dei sottoproblemi

Per come sono stati definiti i sottoproblemi, ciascun sottoproblema corrisponde ad una cella della matrice triangolare superiore sinistra. A tal proposito il numero di sottoproblemi è $\frac{n^2}{2} = O(n^2)$.

Goal

Il valore dell'albero minimo che stiamo cercando si trova all'interno della cella $OPT[n, 1].val$.
Questa cella contiene il valore dell'albero minimo che include tutte le foglie che vanno da $C[1]$ a $C[n]$ (essendo $i = 1$ e $j = n$). Poiché questo albero contiene tutte le foglie, è l'albero di costo minimo che stiamo cercando.

2.4 Formula di Bellman

Caso base, $i = 1$

$$OPT[i,j] \begin{cases} .val = 0 \\ .max = C[j] \\ .sin = NULL \\ .des = NULL \end{cases}$$

Caso base, $i = 2$

$$OPT[i,j] \begin{cases} .val = OPT[i-1,j].max \times OPT[i-1,j+1].max \\ .max = \max\{OPT[i-1,j].max, OPT[i-1,j+1].max\} \\ .sin = (i-1,j) \\ .des = (i-1,j+1) \end{cases}$$

Caso generico

$$OPT[i,j].val = \min_{k=1,\ \ldots,\ i-1}\{(OPT[k,j].max \times OPT[i-k,j+k].max) + OPT[k,j].val + OPT[i-k,j+k].val\}$$

Dopo aver trovato il k che minimizza la formula sopra, calcoliamo i restanti 3 campi di $OPT[i,j]$

$$OPT[i,j] \begin{cases} .max = \max\{(OPT[k,j].max, OPT[i-k,j+k].max)\} \\ .sin = (k,j) \\ .des = (i-k,j+k) \end{cases}$$

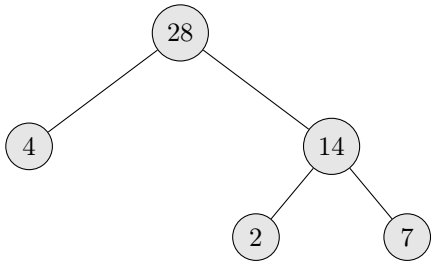
Esempio

Supponiamo di avere in input il seguente vettore $C = [4, 2, 7]$ con 3 foglie.

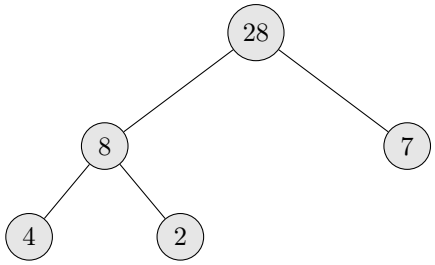
Ogni cella della matrice contiene 4 valori, dunque possiamo codificare questi 4 valori come il seguente array: $[.val, .max, .sin, .des$

$$\begin{bmatrix} [0,4,-,-] & [0,2,-,-] & [0,7,-,-] \\ [8,4,(1,1),(1,2)] & [7,14,(1,2),(1,3)] & [-,-,-,-] \\ [?,?,?,?] & [-,-,-,-] & [-,-,-,-] \end{bmatrix}$$

Per riempire la cella $OPT[3, 1]$, che contiene il valore dell'albero minimo, dobbiamo determinare la combinazione ottimale delle foglie che produce l'albero di costo minimo. Quindi, dobbiamo trovare l'indice k compreso tra 1 e $i - 1$. Considerando la cella $(3, 1)$, dove $i = 3$, i possibili valori per k sono 1 e 2.



Con $k = 1$ dobbiamo combinare i valori contenuti in $OPT[1, 1]$ e $OPT[2, 2]$, portando così il costo dell'allbero a 42



Con $k = 2$ dobbiamo combinare i valori contenuti in $OPT[2,1]$ e $OPT[1,3]$, portando così il costo dell'allbero a 36

$$\begin{bmatrix} [0,4,-,-] & [0,2,-,-] & [0,7,-,-] \\ [8,4,(1,1),(1,2)] & [7,14,(1,2),(1,3)] & [-,-,-,-] \\ [36,7,(2,1),(1,3)] & [-,-,-,-] & [-,-,-,-] \end{bmatrix}$$

2.5 Pseudocodice

Algorithm 2 Algoritmo_Albero

```
1: function COMPUTE_OPT(C[1: n])
2:     Matrix OPT[n, n]
3:                                                          ▷ Caso base, i = 1
4:     for j = 1 to n do
5:         OPT[1, j].val = 0
6:         OPT[1, j].max = C[j]
7:         OPT[1, j].sin = NULL
8:         OPT[1, j].des = NULL
9:     end for
10:                                                         ▷ Caso base, i = 2
11:    if n >= 2 then
12:        for j = 1 to n do
13:            OPT[2, j].val = OPT[i − 1, j].max × OPT[i − 1, j + 1].max
14:            OPT[2, j].max = max{OPT[i − 1, j].max, OPT[i − 1, j + 1].max}
15:            OPT[2, j].sin = (i − 1, j)
16:            OPT[2, j].des = (i − 1, j + 1)
17:        end for
18:    end if
19:                                                         ▷ Fine casi base
20:                                                         ▷ Casi Generici
21:    if n >= 3 then
22:        for i = 1 to n do
23:            for j = 1 to n − i + 1 do
24:                OPT[i, j].val = min_{k=1, ..., i−1}{(OPT[k, j].max × OPT[i − k, j + k].max) + OPT[k, j].val + OPT[i −
   k, j + k].val}
25:                Salviamo il k che minimizza OPT[i, j].val
26:                OPT[i, j].max = max{(OPT[k, j].max, OPT[i − k, j + k].max)}
27:                OPT[i, j].sin = (k, j)
28:                OPT[i, j].des = (i − k, j + k)
29:            end for
30:        end for
31:    end if
       return OPT[n, 1].val
32: end function
```

Analisi della complessità

- Il tempo di esecuzione dell'algoritmo è una diretta conseguenza del numero dei sottoproblemi. Infatti avendo $O(n^2)$ sottoproblemi, ciascuno risolvibile in tempo $O(k)$, segue che l'algoritmo trova la soluzione ottima in tempo $O(n^2 \times k)$, dove $k = O(n)$. In conlusione il tempo di esecuzione è $O(n^3)$.
  Siamo consapevoli che la complessità temporale di questo algoritmo non sia ottimale.

- Per quanto riguarda lo spazio utilizzato, usando la tecnica tabulation, in memoria salviamo una matrice $n \times n$ dove ciascuna cella contiene 4 valori. Di conseguenza, l'algoritmo calcola la soluzione ottima in spazio $4n^2 = O(n^2)$.

2.6 Ricostruzione dell'albero

Per quanto concerne la ricostruzione dell'albero, a partire dalla matrice OPT, in tempo lineare nel numero di nodi dell'albero possiamo ricostruire l'albero nel seguente modo:

- Partiamo dalla cella $OPT[n, 1]$, che contiene la radice dell'albero, in quanto contiene la somma dei valori dei nodi interni dell'albero.
- Il valore di ciascun nodo lo ricaviamo guardando le coppie degli indici $.sin$ e $.des$.
  Definiamo $sin_{i,j} = OPT[i, j].sin$ e $des_{i,j} = OPT[i, j].des$.
  Dunque, il valore della radice è:
  $root = OPT[n, 1].val - (OPT[sin_{n,1}].val + OPT[des_{n,1}].val)$
- In generale, per un nodo $x_{i,j}$, $val(x_{i,j}) = OPT[i, j].val - (OPT[sin_{i,j}].val + OPT[des_{i,j}.val])$

Per comodità, assumiamo $sin = (sin_0,\ sin_1)$ e $des = (des_0,\ des_1)$, quindi

$$OPT[sin] = OPT[sin_0,\ sin_1]$$

$$OPT[des] = OPT[des_0,\ des_1]$$

Algorithm 3 RicostruzioneAlbero(OPT)

```
1:  function RicostruzioneAlbero(OPT)
2:      T ← albero vuoto
3:      sin ← OPT[n, 1].sin
4:      des ← OPT[n, 1].des
5:
6:      if sin = NULL and des = NULL then                          ▷ foglia
7:          T.root ← OPT[i, j].max
8:          return T
9:      end if
10:
11:     T.root ← OPT[n, 1].val − (OPT[sin].val + OPT[des].val)
12:     RicostruzioneAlberoRicorsivo(OPT, T, sin[0], sin[1], root)
13:     RicostruzioneAlberoRicorsivo(OPT, T, des[0], des[1], root)
14:
15:     return T
16: end function
17:
18:
19: function RicostruzioneAlberoRicorsivo(OPT, T, i, j, parent)
20:     sin ← OPT[i, j].sin
21:     des ← OPT[i, j].des
22:
23:     if sin = NULL and des = NULL then                          ▷ foglia
24:         val ← OPT[i, j].max
25:         return
26:     end if
27:
28:     v.val ← OPT[i, j].val − (OPT[sin].val + OPT[des].val)
29:     rendi v figlio di parent in T
30:
31:     RicostruzioneAlberoRicorsivo(OPT, T, sin[0], sin[1], v)
32:     RicostruzioneAlberoRicorsivo(OPT, T, des[0], des[1], v)
33: end function
```