Struttura Dati Succinta per la Gestione di Sequenze di Parentesi ben Bilanciate

Fabrizio Brioni

2 giugno 2022

Sommario

In questo documento verrà mostrata una struttura succinta per memorizzare ed effettuare operazioni su una sequenza di parentesi ben bilanciata. Inizialmente verrà descritta la struttura dati segment tree, dopodiché vi sarà la descrizione di un'organizzazione a due livelli (simile alla struttura di jacobson per il rank) per la memorizzazione di informazioni riguardanti le sequenze di parentesi ben bilanciate e infine una spiegazione di come integrare le due strutture per gestire operazioni di tipo find_close, find_open, find_enclose in modo efficiente e succinto.

1 Introduzione

Una sequenza V di parentesi è definita ben bilanciata se in ciascun prefisso di V il numero di parentesi aperte è maggiore o uguale al numero di parentesi chiuse e il numero complessivo di parentesi aperte in V è uguale al numero di parentesi chiuse. In termini meno formali, può essere definita come una sequenza di parentesi da cui è possibile ottenere un'operazione aritmetica valida inserendo eventuali numeri e operazioni. Alcune operazioni utili eseguibili su tali sequenze sono le sequenti:

- `find_close(i)`: data una parentesi aperta in posizione i, calcolare la posizione delle corrispettiva parentesi chiusa;
- `find_open(i)`: data una parentesi chiusa in posizione i, calcolare la posizione delle corrispettiva parentesi aperta;
- `find_enclose(i)`: data una parentesi in posizione i, calcolare le posizioni (x, y) della coppia di parentesi corrispondenti che la racchiudono. Nel caso vi siano più coppie con tale proprietà, cercare quella più interna, ovvera quella che non contiene altre coppie valide (per una definizione più formale si legga la sezione **Notazioni**).

In questo documento verrà mostrato come, data una sequenza di parentesi ben bilanciate di lunghezza $2n$, eseguire tali operazioni con una complessità temporale di $\mathcal{O}(\log n)$ utilizzando $2n + o(n)$ bits di memoria.

2 Notazioni

Sia v una sequenza di parentesi di lunghezza $2n$ numerate da 0 a $2n-1$ e sia v_i la parentesi di indice i. Per comodità supponiamo che un valore $v_i = 1$ indichi una parentesi aperta in posizione i e un valore $v_i = 0$ indichi una parentesi chiusa in posizione i. Definiamo l'eccesso di una posizione i come:

$$\text{excess}_v(i) = |\{j : \ j < i \wedge v_i = 1\}| - |\{j : \ j \leq i \wedge v_i = 0\}|$$

ovvero la differenza tra il numero di parentesi aperte che precedono i (escluso) e il numero di parentesi chiuse che precedono i (questa volta incluso). Ne segue che una sequenza di parentesi è ben bilanciata se e solo se:

$$\begin{aligned} \text{excess}_v(i) &\geq 0 \qquad 0 \leq i < 2n-1 \\ \text{excess}_v(2n-1) &= 0 \end{aligned}$$

Inoltre ne consegue la seguente definizione delle operazioni di find_close, find_open e find_enclose:

$$
\begin{aligned}
\text{find\_close}_v(i) &= \min\{j : \ j > i \wedge \text{excess}_v(j) = \text{excess}_v(i)\}\\
\text{find\_open}_v(i) &= \max\{j : \ j < i \wedge \text{excess}_v(j) = \text{excess}_v(i)\}\\
\text{left\_enclose}_v(i) &= \max\{j : \ j < i \wedge \text{excess}_v(j) + 1 = \text{excess}_v(i)\}\\
\text{right\_enclose}_v(i) &= \text{find\_close}_v(\text{left\_enclose}_v(i)) = \min\{j : \ j > i \wedge \text{excess}_v(j) + 1 = \text{excess}_v(i)\}\\
\text{find\_enclose}_v(i) &= (\text{left\_enclose}_v(i), \text{right\_enclose}_v(i))
\end{aligned}
$$

Inoltre useremo t^v per indicare una sequenza di interi di lunghezza $2n$ tale che:

$$t_i^v = \text{excess}_v(i) \qquad 0 \leq i < 2n - 1$$

Infine definiamo una funzione `find_succ`(x, i, v) che data una sequenza $x_0, x_1, \ldots$ di interi e due interi i e v restituisca l'indice del primo elemento che segue x_i ed ha valore minore o uguale a v:

$$\text{find\_succ}(x, i, v) = \min\{j : \ j > i \wedge x_j \leq v\}$$

in modo analogo definiamo una funzione `find_prev`(x, i, v) che restituisca l'indice dell'ultimo elemento che precede x_i ed ha valore minore o uguale a v:

$$\text{find\_prev}(x, i, v) = \max\{j : \ j < i \wedge x_j \leq v\}$$

3 Segment Tree

Il Segment Tree relativo a una sequenza $x_0, x_1, \ldots, x_{n-1}$ di n valori è un albero binario che ha come radice un nodo contenente informazioni relative all'intera sequenza (come ad esempio la somma, il valore massimo o il minimo) e (se $n \neq 1$) avente come sottoalbero sinistro un Segment Tree relativo alla sequenza $x_0, \ldots, x_{\lfloor \frac{n-1}{2} \rfloor}$ e come sottoalbero destro un SegmentTree relativo alla sequenza $x_{\lfloor \frac{n-1}{2} \rfloor + 1}, \ldots, x_{n-1}$. Dato un nodo k di tale albero, con $k.data$ indicheremo l'informazione contenuta (nel nostro caso sarà sempre contenuto il minimo dell'intervallo di competenza) e con $k.left$ e $k.right$ rispettivamente il figlio sinistro e destro.

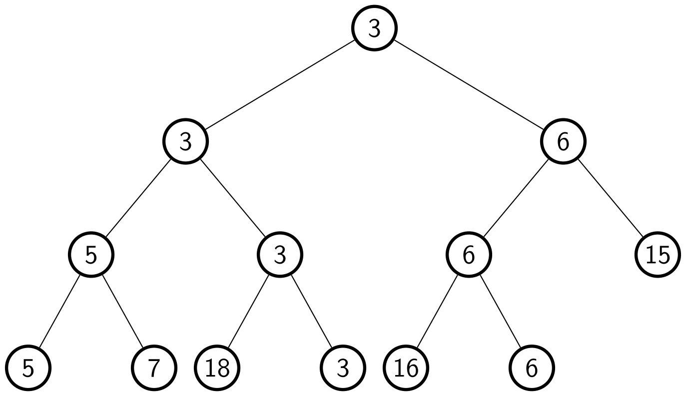

Figura 1: Segment Tree relativo alla sequenza $5, 7, 18, 3, 16, 6, 15$. In ciascun nodo è contenuto il valore minimo della sequenza di cui si occupa.

3.1 Costruzione

La seguente procedura restituisce la radice del Segment Tree relativo alla sequenza x di lunghezza $n = size(x)$:

procedure BUILD(x)
    $root \leftarrow$RECURSIVE_BUILD$(x, 0, size(x) - 1)$
    return $root$
end procedure

function RECURSIVE_BUILD(x, l, r)
    $tmp \leftarrow$ new Segment Tree node
    if $l = r$ **then**
        $tmp.data \leftarrow x_l$

$tmp.left \leftarrow null$
$tmp.right \leftarrow null$
else
$tmp.left \leftarrow$ Recursive_build$(x, l, \lfloor \frac{l+r}{2} \rfloor)$
$tmp.right \leftarrow$ Recursive_build$(x, \lfloor \frac{l+r}{2} \rfloor + 1, r)$
$tmp.data \leftarrow \min\{tmp.left.data, tmp.right.data\}$
end if
return tmp
end function

Per una sequenza lunga n il numero $f(n)$ di nodi creati è:

$$f(n) = \begin{cases} 1 & n = 1 \\ 1 + f(\lfloor \frac{n}{2} \rfloor) + f(\lceil \frac{n}{2} \rceil) & n > 1 \end{cases}$$

ne consegue che $f(n) = \mathcal{O}(n)$ e poiché la funzione `Recursive_build` viene chiamata una volta per ciascun nodo, la complessità della procedura di costruzione è $\mathcal{O}(n)$. Inoltre supponendo che ciascun elemento delle sequenza abbia un valore compreso fra 0 e A, il numero di bit necessari per contenere le informazioni di tutti i nodi è $\mathcal{O}(n \log A)$.

3.2 find_succ e find_prev

Grazie a tale struttura è possibile implementare in modo efficiente la funzione find_succ(x, i, v) una volta costruito il Segment Tree relativo ad x la cui radice è `root`:

procedure Find_succ$(root, x, i, v)$
return Find_succ_recursive$(root, i, v, 0, size(x) - 1)$
end procedure

function Find_succ_recursive$(node, ind, val, l, r)$
if $ind \geq r \vee node.data > val$ **then**
$res \leftarrow \infty$
else if $l = r$ **then**
$res \leftarrow l$
else
$res \leftarrow$ Find_succ_recursive$(node.left, ind, val, l, \lfloor \frac{l+r}{2} \rfloor)$
if $res \neq \infty$ **then**
$res \leftarrow$ Find_succ_recursive$(node.right, ind, val, \lfloor \frac{l+r}{2} \rfloor + 1, r)$
end if
end if
return res
end function

Il numero di nodi visitati e la complessità di tale procedura è $\mathcal{O}(\log n)$. In modo del tutto analogo è possibile implementare la funzione find_prev(x, i, v):

procedure Find_prev$(root, x, i, v)$
return Find_prev_recursive$(root, i, v, 0, size(x) - 1)$
end procedure

function Find_prev_recursive$(node, ind, val, l, r)$
if $ind \leq l \vee node.data > val$ **then**
$res \leftarrow -\infty$
else if $l = r$ **then**
$res \leftarrow l$
else
$res \leftarrow$ Find_prev_recursive$(node.right, ind, val, \lfloor \frac{l+r}{2} \rfloor + 1, r)$
if $res \neq -\infty$ **then**
$res \leftarrow$ Find_prev_recursive$(node.left, ind, val, l, \lfloor \frac{l+r}{2} \rfloor)$
end if
end if
return res
end function

4 Organizzazione a due livelli delle informazioni

Data una sequenza v di parentesi ben bilanciate di lunghezza $2n$, la si suddivida in *super blocchi* di dimensione $\log^2 2n$ (primo livello) e si suddivida ciascun super blocco in *blocchi* di dimensione $\log 2n$ (secondo livello). Si ponga per comodità $\log 2n = k$.

4.1 Gestione Primo Livello

Il numero di super blocchi in questo livello è $\frac{2n}{k^2}$ (numerati da 0 a $\frac{2n}{k^2} - 1$), per ognuno di essi si calcoli l'eccesso minimo presente ottenendo la sequenza S:

$$S_i = \min_{ik^2 \leq j < (i+1)k^2} \{t_j^v\} \qquad 0 \leq i < \frac{2n}{k^2}$$

Si costruisca una Segment Tree relativo a questa sequenza: poiché gli elementi di t^v sono compresi tra 0 e $2n$ (poiché sono gli eccessi di una sequenza di $2n$ parentesi) e la sequenza S è lunga $\frac{2n}{k^2}$, tale Segment Tree occuperà un numero di bit pari a $\mathcal{O}(\frac{2n}{k^2} \log 2n) = \mathcal{O}(\frac{2n}{\log 2n}) = o(n)$.

Inoltre si calcoli la sequenza T composta dall'eccesso iniziale di ciascun super blocco:

$$T_i = t_{ik^2}^v \qquad 0 \leq i < \frac{2n}{k^2}$$

salvando la sequenza T in un array il numero di bit necessari per memorizzarla sarà anch'esso $\frac{2n}{k^2} \log 2n = o(n)$, quindi in totale il numero di bit utilizzati per salvare in memoria questo primo livello di informazioni è $o(n)$.

4.2 Gestione Secondo Livello

Il numero di blocchi in questo livello è $\frac{2n}{k}$ (numerati da 0 a $\frac{2n}{k} - 1$), per ognuno di essi si calcoli l'eccesso minimo presente come differenza rispetto all'eccesso iniziale del relativo super blocco:

$$B_i = \left(\min_{ik \leq j < (i+1)k} \{t_j^v\} \right) - T_{\lfloor \frac{i}{k} \rfloor} \qquad 0 \leq i < \frac{2n}{k}$$

salvando la sequenza B in un array il numero di bit necessari per memorizzarla è $\frac{2n}{k} \log\left(2(\log 2n)^2\right) = \frac{2n}{\log 2n}(2 \log\log 2n + \log 2) = o(n)$ perchè ciascun elemento di B sarà compreso tra $-(\log 2n)^2$ e $(\log 2n)^2$.

Inoltre si calcoli la sequenza A composta dalla differenza tra l'eccesso iniziale di ciascun blocco e l'eccesso del relativo super blocco:

$$A_i = t_{ik}^v - T_{\lfloor \frac{i}{k} \rfloor} \qquad 0 \leq i < \frac{2n}{k}$$

salvando la sequenza A in un array il numero di bit necessari per memorizzarla è $\frac{2n}{k} \log\left(2(\log 2n)^2\right) = o(n)$, quindi in totale il numero di bit utilizzati per salvare in memoria questo secondo livello di informazioni è $o(n)$.

5 find_close e find_open

Una volta creato il Segment Tree relativo a S (la cui radice verrà indicata con `root`) e calcolati i valori di T, B e A relativi alla sequenza di parentesi v è possibile eseguire l'operazione `find_close(i)` (supponendo che la parentesi in posizione i sia aperta) nel seguente modo:

1. Si calcola l'eccesso $u = t_i^v = T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} + (t_i^v - t_{k(\lfloor \frac{i}{k} \rfloor)}^v) = T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} + 2\sum_{j=k(\lfloor \frac{i}{k} \rfloor)}^{i-1} v_j - (i - k(\lfloor \frac{i}{k} \rfloor)) + (1 - v_{k(\lfloor \frac{i}{k} \rfloor)})$ che è possibile calcolare in $\mathcal{O}(\log n)$ (la sommatoria contiene al più $\log 2n$ addendi);

2. Si verifica se la posizione cercata appartiene allo stesso blocco dell'indice i effettuando una scansione lineare di tutte le parentesi che seguono i fino alla fine del blocco a cui esso appartiene $(v_{i+1}, v_{i+2}, \ldots, v_{k(\lfloor \frac{i}{k} \rfloor + 1) - 1})$: se vi è un indice x tale che il numero di parentesi aperte e chiuse comprese fra i e x è uguale allora tale indice x è la posizione cercata. Durante questo processo viene fatto accesso al più $k = \log 2n$ volte alla sequenza v;

3. Altrimenti si verifica se la posizione cercata appartiene allo stesso super blocco dell'indice i effettuando una scansione dei valori di B relativi ai blocchi successivi a quello di i fino alla fine del super blocco a cui appartiene i ($B_{\lfloor \frac{i}{k} \rfloor + 1}, \ldots, B_{k(\lfloor \frac{i}{k^2} \rfloor + 1) - 1}$): se vi è un indice x tale che $B_x + T_{\lfloor \frac{i}{k^2} \rfloor} \leq t_i^v$ allora la posizione cercata si trova nel blocco x (e questo blocco si trova nello stesso super blocco di i), in tal caso per trovare la posizione esatta basta una scansione di tutte le parentesi di tale blocco ($v_{xk}, v_{xk+1}, \ldots, v_{xk+k-1}$) finchè non si trova il primo indice y in cui vale $t_y^v = t_i^v$ (si noti come sia possibile calcolare mano a mano il valore di t_y^v utilizzando la stessa formula del punto 1). Durante questo processo viene fatto accesso al più $k = \log 2n$ volte alla sequenza B e al più $k = \log 2n$ volte alla sequenza v;

4. Se non è stata trovata la posizione dopo gli step 2 e 3 significa che la posizione cercata si trova in un altro super blocco rispetto a quello in cui si trova i, per trovare tale super blocco basterà chiamare la procedura `Find_succ`$(B, \lfloor \frac{i}{k^2} \rfloor, t_i^v)$ che provvederà a ritornare l'indice x del primo super blocco che segue i contenente un eccesso minore o uguale a t_i^v (e poichè due valori consecutivi t^v differiscono di al più 1, quel super blocco conterrà sicuramente un eccesso uguale a t_i^v). A quel punto va ricercato il blocco in cui si trova la posizione cercata e dopodiché si ricerca la posizione esatta in modo analogo a quanto spiegato nello step 3: si effettua una scansione dei valori di B relativi ai blocchi contenuti nel super blocco x ($B_{xk}, \ldots, B_{xk+k-1}$) e se vi è un indice y tale che $B_y + T_x \leq t_i^v$ allora la posizione cercata si trova nel blocco y, infine per trovare la posizione esatta basta una scansione di tutte le parentesi di tale blocco ($v_{yk}, \ldots, v_{yk+k-1}$) finchè non si trova il primo indice z in cui vale $t_z^v = t_i^v$. Durante questo processo viene effettuata una chiamata alla funzione `Find_succ` (che ha complessità $\mathcal{O}(\log \frac{2n}{k^2}) = \mathcal{O}(\log n)$) e viene fatto accesso al più $k = \log 2n$ volta alla sequenza B e al più $k = \log 2n$ volta alla sequenza v.

In conclusione è possibile effettuare l'operazione `find_close` con complessità $\mathcal{O}(\log n)$ utilizzando la sequenza v di parentesi che occupa $2n$ bit e altre strutture ausiliare (il Segment Tree relativo alla sequenza S e le sequenze T, B, A) che occupano $o(n)$ bit. Il seguente pseudocodice mostra l'implementazione di tutta la procedura:

Algorithm 1 `Find_close`

procedure FIND_CLOSE($root, T, A, B, n, v, i$) ▷ si suppone v_i parentesi aperta
    $k \leftarrow \log 2n$
    $u \leftarrow T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} - (i - k(\lfloor \frac{i}{k} \rfloor))$ ▷ step 1
    for $j \leftarrow k(\lfloor \frac{i}{k} \rfloor)$ **to** $i - 1$ **do**
        $u \leftarrow u + 2v_j$
    end for
    $u \leftarrow u + (1 - v_{k(\lfloor \frac{i}{k} \rfloor)})$

    $tmp \leftarrow 0$ ▷ step 2
    for $x \leftarrow i$ **to** $k(\lfloor \frac{i}{k} \rfloor + 1) - 1$ **do**
        if $v_x = 1$ **then**
            $tmp \leftarrow tmp + 1$
        else
            $tmp \leftarrow tmp - 1$
        end if
        if $tmp = 0$ **then**
            return x
        end if
    end for

    for $x \leftarrow \lfloor \frac{i}{k} \rfloor + 1$ **to** $k(\lfloor \frac{i}{k^2} \rfloor + 1) - 1$ **do** ▷ step 3
        if $B_x + T_{\lfloor \frac{i}{k^2} \rfloor} \leq u$ **then**
            $currt \leftarrow T_{\lfloor \frac{i}{k^2} \rfloor} + A_x$
            for $y \leftarrow xk$ **to** $xk + k - 1$ **do**
                if $currt = u$ **then**
                    return y
                end if

```
                if v_y = 1 then
                    currt ← currt + 1
                end if
                if v_{y+1} = 0 then
                    currt ← currt − 1
                end if
            end for
        end if
    end for

    x ← Find_succ(root, B, ⌊i/k²⌋, u)                                ▷ step 4
    for y ← xk to xk + k − 1 do
        if B_y + T_x ≤ u then
            currt ← T_x + A_y
            for z ← yk to yk + k − 1 do
                if currt = u then
                    return z
                end if
                if v_z = 1 then
                    currt ← currt + 1
                end if
                if v_{z+1} = 0 then
                    currt ← currt − 1
                end if
            end for
        end if
    end for
end procedure
```

In modo simmetrico è possibile implementare l'operazione `find_open(i)`: basti notare che l'operazione `find_open(i)` relativa ad una sequenza $v = v_0, \ldots, v_{2n-1}$ è equivalente a un'operazione di `find_close(2n-1-i)` relativa alla sequenza sequenza $v_{2n-1}, \ldots, v_0$. Inizialmente si verifica se la posizione cercata è presente nello stesso blocco di i, altrimenti si verifica se è presente nello stesso super blocco di i e in caso contrario si va a ricercare il super blocco di appartenenza, il blocco corretto all'interno del super blocco e infine la posizione esatta all'interno del blocco:

Algorithm 2 `Find_open`

```
procedure Find_open(root, T, A, B, n, v, i)              ▷ si suppone v_i parentesi chiusa
    k ← log 2n
    u ← T_{⌊i/k²⌋} + A_{⌊i/k⌋} − (i − k(⌊i/k⌋) + 1)                   ▷ step 1
    for j ← k(⌊i/k⌋) to i − 1 do
        u ← u + 2v_j
    end for
    u ← u + (1 − v_{k(⌊i/k⌋)})

    tmp ← 0                                                          ▷ step 2
    for x ← i down to k⌊i/k⌋ do
        if v_x = 1 then
            tmp ← tmp + 1
        else
            tmp ← tmp − 1
        end if
        if tmp = 0 then
            return x
        end if
    end for
```

for $x \leftarrow \lfloor \frac{i}{k} \rfloor - 1$ **down to** $k(\lfloor \frac{i}{k^2} \rfloor)$ **do** ▷ step 3
    if $B_x + T_{\lfloor \frac{i}{k^2} \rfloor} \leq u$ **then**
        $currt \leftarrow T_{\lfloor \frac{i}{k^2} \rfloor} + A_{x+1}$
        for $y \leftarrow xk + k - 1$ **down to** xk **do**
            if $v_{y+1} = 0$ **then**
                $currt \leftarrow currt + 1$
            end if
            if $v_y = 1$ **then**
                $currt \leftarrow currt - 1$
            end if
            if $currt = u$ **then**
                return y
            end if
        end for
    end if
end for

$x \leftarrow$ FIND_PREV$(root, B, \lfloor \frac{i}{k^2} \rfloor, u)$ ▷ step 4
for $y \leftarrow xk + k - 1$ **down to** xk **do**
    if $B_y + T_x \leq u$ **then**
        if $y < xk + k - 1$ **then**
            $currt \leftarrow T_x + A_{y+1}$
        else
            $currt \leftarrow T_{x+1}$
        end if
        for $z \leftarrow yk + k - 1$ **down to** yk **do**
            if $v_{z+1} = 0$ **then**
                $currt \leftarrow currt + 1$
            end if
            if $v_z = 1$ **then**
                $currt \leftarrow currt - 1$
            end if
            if $currt = u$ **then**
                return z
            end if
        end for
    end if
end for
end procedure

6 Find_enclose

L'operazione `left_enclose(i)` è analoga a `find_open(i)` con l'unica differenza che l'eccesso cercato non è pari a t_i^v ma bensì $t_i^v - 1$ (e che non esiste una parentesi con tale eccesso se $t_i^v = 0$):

Algorithm 3 `Left_enclose`

procedure LEFT_ENCLOSE$(root, T, A, B, n, v, i)$
    $k \leftarrow \log 2n$
    $u \leftarrow T_{\lfloor \frac{i}{k^2} \rfloor} + A_{\lfloor \frac{i}{k} \rfloor} - (i - k(\lfloor \frac{i}{k} \rfloor))$ ▷ step 1
    for $j \leftarrow k(\lfloor \frac{i}{k} \rfloor)$ **to** $i - 1$ **do**
        $u \leftarrow u + 2v_j$
    end for
    $u \leftarrow u + (1 - v_{k(\lfloor \frac{i}{k} \rfloor)})$
    if $v_i = 0$ **then**
        $u \leftarrow u - 1$
    end if
    $u \leftarrow u - 1$ ▷ l'eccesso cercato ora è $t_i^v - 1$

```
    if u=-1 then
        return −1                ▷ Non esiste una coppia di parentesi che racchiude l'indice i
    end if

    tmp ← u + 1                                                      ▷ step 2
    for x ← i down to k⌊i/k⌋ do
        if tmp = u then
            return x
        end if
        if v_x = 0 then
            tmp ← tmp + 1
        end if
        if v_{x−1} = 1 then
            tmp ← tmp − 1
        end if
    end for

    for x ← ⌊i/k⌋ − 1 down to k(⌊i/k^2⌋) do                          ▷ step 3
        if B_x + T_{⌊i/k^2⌋} ≤ u then
            currt ← T_{⌊i/k^2⌋} + A_{x+1}
            for y ← xk + k − 1 down to xk do
                if v_{y+1} = 0 then
                    currt ← currt + 1
                end if
                if v_y = 1 then
                    currt ← currt − 1
                end if
                if currt = u then
                    return y
                end if
            end for
        end if
    end for

    x ← FIND_PREV(root, B, ⌊i/k^2⌋, u)                               ▷ step 4
    for y ← xk + k − 1 down to xk do
        if B_y + T_x ≤ u then
            if y < xk + k − 1 then
                currt ← T_x + A_{y+1}
            else
                currt ← T_{x+1}
            end if
            for z ← yk + k − 1 down to yk do
                if v_{z+1} = 0 then
                    currt ← currt + 1
                end if
                if v_z = 1 then
                    currt ← currt − 1
                end if
                if currt = u then
                    return z
                end if
            end for
        end if
    end for
end procedure
```

E di conseguenza l'operazione `find_enclose` non è nient'altro che una chiamata a `left_enclose` e `find_close`:

Algorithm 4 `Find_enclose`

```
procedure FIND_ENCLOSE(root, T, A, B, n, v, i)
    x ← LEFT_ENCLOSE(root, T, A, B, n, v, i)
    if x = −1 then
        return (−1, −1)                                ▷ non esiste soluzione
    else
        return (x, FIND_CLOSE(x))
    end if
end procedure
```

7 Conclusioni

È stato mostrato come sia possibile effettuare le operazioni `find_close`, `find_open` e `find_enclose` con complessità $\mathcal{O}(\log n)$ utilizzando $2n + o(n)$ bits. Inoltre la costruzione iniziale di tutte le strutture necessarie ha complessità $\mathcal{O}(n)$.