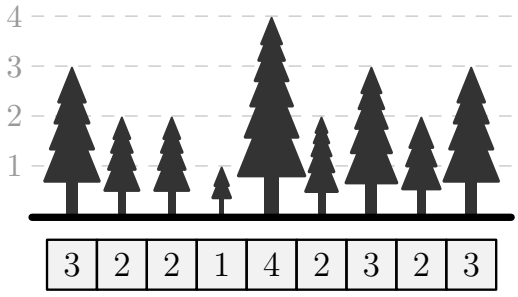
Taglialegna (`taglialegna`)

Limite di tempo:	1.0 secondi
Limite di memoria:	256 MiB

La Abbatti S.p.A. è una grossa azienda che lavora nel settore del disboscamento. In particolare, nel tempo si è specializzata nel taglio degli *alberi cortecciosi*, una tipologia di alberi estremamente alti, robusti e ostinati. Si tratta di una specie molto ordinata: i boschi formati da questi alberi consistono in una lunghissima fila di tronchi disposti lungo una fila orizzontale a esattamente un decametro l'uno dall'altro. Ogni albero ha una altezza, espressa da un numero (positivo) di decametri.



(i valori rappresentano le altezze in decametri)

Il taglio di un albero corteccioso è un compito delicato e, nonostante l'uso delle più avanzate tecnologie di abbattimento, richiede comunque molto tempo, data la loro cortecciosità. Gli operai sono in grado di segare i tronchi in modo che l'albero cada a destra o a sinistra, secondo la loro scelta.

Quando un albero corteccioso viene tagliato e cade, si abbatte sugli eventuali alberi non ancora tagliati che si trovano nella traiettoria della caduta, ovvero tutti quegli alberi non ancora tagliati che si trovano ad una distanza strettamente minore dell'altezza dell'albero appena segato, nella direzione della caduta. Data la mole degli alberi cortecciosi, gli alberi colpiti dalla caduta vengono a lora volta spezzati alla base, cadendo nella direzione dell'urto, innescando un effetto domino.

Per assicurarsi il primato nel settore, la Abbatti S.p.A. ha deciso di installare un sistema in grado di analizzare il bosco, determinando quali alberi gli operai dovranno segare, nonchè la direzione della loro caduta, affinchè tutti gli alberi cortecciosi risultino abbattuti alla fine del processo. Naturalmente, il numero di alberi da far tagliare agli operai deve essere il minore possibile, per contenere i costi. In quanto consulente informatico della società, sei incaricato di implementare il sistema.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

- **Subtask 1** [**5 punti**]: Casi d'esempio.
- **Subtask 2** [**9 punti**]: Gli alberi possono essere alti solo 1 o 2 decametri.
- **Subtask 3** [**20 punti**]: $N \leq 50$.
- **Subtask 4** [**19 punti**]: $N \leq 400$.
- **Subtask 5** [**22 punti**]: $N \leq 5000$.
- **Subtask 6** [**14 punti**]: $N \leq 100\,000$.
- **Subtask 7** [**11 punti**]: Nessuna limitazione specifica (vedi la sezione **Assunzioni**).

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

> ☞ Tra gli allegati a questo task troverai un template (`taglialegna.c`, `taglialegna.cpp`, `taglialegna.pas`) con un esempio di implementazione.

Dovrai implementare la seguente funzione:

C/C++	`void Pianifica(int N, int altezza[]);`
Pascal	`procedure Pianifica(N: longint; var altezza: array of longint);`

N è il numero di alberi cortecciosi nel bosco, mentre `altezza[i]` contiene, per ogni $0 \leq i < N$, l'altezza, in decametri, dell'i-esimo albero corteccioso a partire da sinistra. La funzione dovrà chiamare la routine già implementata

C/C++	`void Abbatti(int indice, int direzione);`
Pascal	`procedure Abbatti(indice: longint; direzione: longint);`

dove `indice` è l'indice (da 0 a $N-1$) dell'albero da abbattere, e `direzione` è un intero che vale 0 se l'albero deve cadere a sinistra e 1 se invece deve cadere a destra.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file `input.txt`, a quel punto chiama la funzione `Pianifica` che dovete implementare. Il grader scrive sul file `output.txt` il resoconto delle chiamate ad `Abbatti`.

Nel caso vogliate generare un input per un test di valutazione, il file `input.txt` deve avere questo formato:

- Riga 1: contiene l'intero N, il numero di alberi cortecciosi nel bosco (consigliamo di non superare il valore 50 data l'inefficienza del grader fornito).
- Riga 2: contiene N interi, di cui l'i-esimo rappresenta l'altezza in decametri dell'albero di indice i.

Il file `output.txt` invece ha questo formato:

- Righe dalla 1 in poi: La i-esima di queste righe contiene i due parametri passati alla funzione `Abbatti`, cioè l'indice dell'albero tagliato e la direzione della caduta (0 indica sinistra e 1 indica destra), nell'ordine delle chiamate.

Assunzioni

- $1 \leq N \leq 2\,000\,000$.
- L'altezza di ogni albero è un numero intero di decametri compreso tra 1 e $1\,000\,000$.
- Un'esecuzione del programma viene considerata errata se:
  - Al termine della chiamata a `Pianifica` tutti gli alberi sono caduti, ma il numero di alberi segati dagli operai non è il minimo possibile.

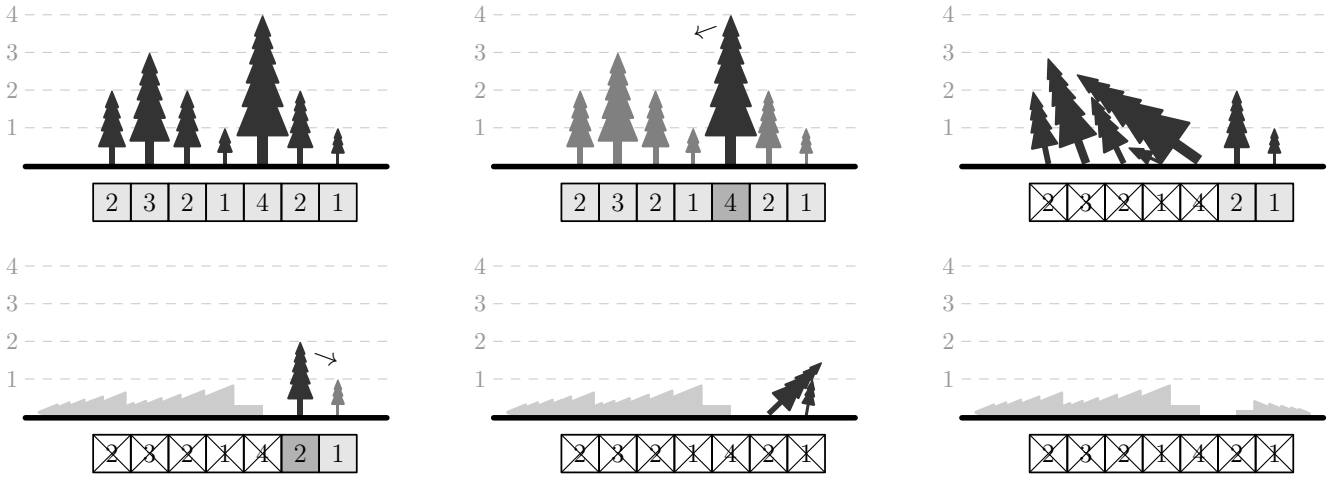
– Al termine della chiamata a `Pianifica` non tutti gli alberi sono caduti.
– Viene fatta una chiamata ad `Abbatti` con un indice o una direzione non validi.
– Viene fatta una chiamata ad `Abbatti` con l'indice di un albero già caduto, direttamente ad opera degli operai o indirettamente a seguito dell'urto con un altro albero.

Esempi di input/output

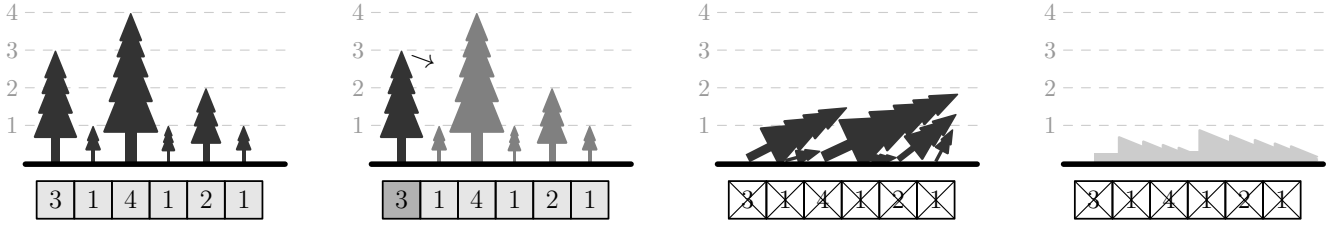
input.txt	output.txt
7 2 3 2 1 4 2 1	4 0 5 1
6 3 1 4 1 2 1	0 1

Spiegazione

Nel **primo caso d'esempio** è possibile abbattere tutti gli alberi segando il quinto albero (alto 4 decametri) facendolo cadere a sinistra, e il sesto albero (alto 2 decametri) facendolo cadere a destra. Il primo albero tagliato innesca un effetto domino che abbatte tutti gli alberi alla sua sinistra, mentre il secondo abbatte l'ultimo albero nella caduta.



Nel **secondo caso d'esempio** tagliando il primo albero in modo che cada verso destra vengono abbattuti anche tutti gli alberi rimanenti.



Soluzione

■ Introduzione e concetti generali

Risolveremo questo problema mediante la tecnica della *programmazione dinamica*. Presenteremo inizialmente una soluzione quadratica nel numero di alberi, e successivamente mostreremo come rendere la stessa soluzione più efficiente.

Entrambe le soluzioni affondano le proprie radici nella stessa osservazione fondamentale, cioè il fatto che possiamo supporre senza perdita di generalità che il primo taglio effettuato dagli operai abbia come effetto quello di abbattere il primo albero. A tal fine, gli operai hanno due opzioni:

- tagliare, facendolo cadere a destra, l'albero 1, oppure
- tagliare, facendolo cadere a sinistra, un albero la cui caduta provochi l'abbattimento dell'albero 1.

In entrambi gli scenari, dopo il primo taglio saranno rimasti intatti solo gli alberi da un certo indice in poi, e ci saremo ridotti a dover abbattere, col minor numero possibile di tagli, un numero minore di alberi rispetto al caso iniziale.

■ Una soluzione $O(N^2)$

Definiamo alcuni concetti che torneranno più volte utili nel corso della spiegazione di entrambe le soluzioni. Chiamiamo `rep` *dell'albero* i, indicato con `rep`$[i]$, l'indice dell'albero più a destra che viene abbattuto dalla caduta dell'albero i, quando questo cade verso destra; analogamente, chiamiamo `lep` *dell'albero* i, indicato con `lep`$[i]$, l'indice dell'albero più a sinistra che viene abbattuto dalla caduta dell'albero i, quando questo cade verso sinistra. Per rendere più chiaro il significato di `lep` e `rep` osserviamo l'esempio in figura 1.

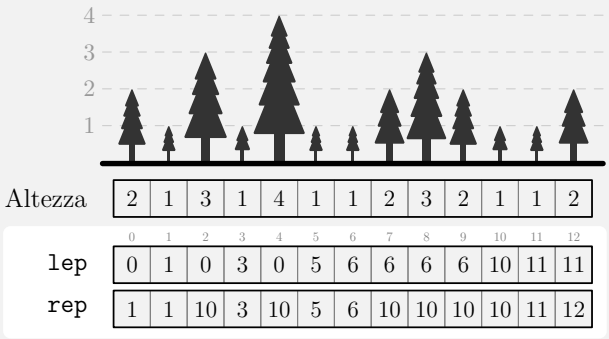


Figura 1

Nell'esempio il `rep` dell'albero 7 è 10, perché l'albero 7, una volta tagliato e lasciato cadere a destra, abbatte l'albero 8, che cade a sua volta abbattendo gli alberi 9 e 10; il `lep` dell'albero 6 invece è 6, perché l'albero in questione è alto 1 decametro e come tale non è in grado di abbattere nessun altro albero.

Calcolare `rep` e `lep` di ogni albero è semplice: il `rep` dell'albero i è pari al maggiore tra i `rep` degli alberi che i abbatte quando cade verso destra, o, nel caso in cui i non abbatta alcun albero nella caduta, a i stesso. Analogamente, il `lep` dell'albero i è pari al minore tra i `lep` degli alberi che i abbatte quando cade verso sinistra, o, nel caso in cui i non abbatta alcun albero nella caduta, a i stesso.

```
1  // Costruzione di rep
2  for (int i = N - 1; i >= 0; i--) {
3      rep[i] = i;
4      for (int j = i; j < i + H[i] && j < N; j++)
5          rep[i] = max(rep[i], rep[j]);
6  }
```

```
1  // Costruzione di lep
2  for (int i = 0; i < N; i++) {
3      lep[i] = i;
4      for (int j = i; j > i - H[i] && j >= 0; j--)
5          lep[i] = min(lep[i], lep[j]);
6  }
```

A questo punto possiamo implementare l'osservazione dell'introduzione, giungendo alla formulazione top-down dell'algoritmo della prossima pagina. Volendo privilegiare la trasparenza nella spiegazione, abbiamo volutamente ignorato la parte di memoizzazione (*memoization*), che non può invece essere trascurata in una implementazione reale.

```cpp
const int INF = numeric_limits<int>::max();
enum direzione_t {SINISTRA, DESTRA};

// Struttura info_t. I significati dei vari membri sono spiegati poco più sotto.
struct info_t {
    int numero_tagli = INF;
    int primo_albero;
    direzione_t direzione;
};

// risolvi(i) ritorna un oggetto info_t, che contiene
//   - numero_tagli: il minimo numero di tagli da effettuare per abbattere tutti gli alberi da
//                   i a N-1 inclusi.
//   - primo_albero: l'indice del primo albero da tagliare
//   - direzione:    la direzione della caduta del primo albero
info_t risolvi(int i) {
    info_t risposta;

    if (i == N) {
        // Se non ci sono alberi da tagliare, numero_tagli = 0
        risposta.numero_tagli = 0;
    } else {
        // Primo caso: abbatti i a destra
        risposta.numero_tagli = risolvi(rep[i] + 1).numero_tagli + 1;
        risposta.primo_albero = i;
        risposta.direzione = DESTRA;

        // Secondo caso: abbatti a sinistra un albero alla destra di i che, nella caduta, abbatta anche i
        for (int j = i; j < N; j++) {
            if (lep[j] <= i) { // Controlla che l'albero j abbatta i cadendo a sinistra
                // Valuta se tagliando l'albero j troviamo una soluzione migliore di quella che conosciamo
                if (risolvi(j + 1).numero_tagli + 1 < risposta.numero_tagli) {
                    risposta.numero_tagli = risolvi(j + 1).numero_tagli + 1;
                    risposta.primo_albero = j;
                    risposta.direzione = SINISTRA;
                }
            }
        }
    }

    return risposta;
}
```

Sfruttando le informazioni calcolate da `risolvi`, è facile ricostruire la sequenza completa dei tagli e risolvere il problema. La soluzione del problema coincide con la chiamata `ricostruisci_tagli(0)` nel codice qui sotto.

```cpp
// ricostruisci_tagli(i) si occupa di tagliare, attraverso opportune chiamate alla funzione Abbatti, il
// minimo numero di alberi affinchè tutti gli alberi da i a N-1 inclusi risultino abbattuti alla fine del
// processo. Riusa internamente le informazioni calcolate dalla funzione risolvi illustrata poco prima
void ricostruisci_tagli(int i) {
    if (i == N)
        return;

    int primo_albero = risolvi(i).primo_albero;
    direzione_t direzione = risolvi(i).direzione;

    Abbatti(primo_albero, direzione);

    if (direzione == SINISTRA)
        ricostruisci_tagli(primo_albero + 1);
    else
        ricostruisci_tagli(rep[i] + 1);
}
```

Analizziamo ora la complessità computazionale dell'algoritmo proposto. Il calcolo dei valori `lep` e `rep` richiede, per ogni albero, al più $O(N)$ operazioni, dunque il numero di operazioni necessarie per calcolare questi valori per tutti gli N alberi è proporzionale N^2. Analogamente, per calcolare `risolvi`(i) è necessario, al caso peggiore, considerare tutti gli alberi alla destra di i, rendendo il calcolo degli N valori di `risolvi`(i) quadratico nel numero di alberi. Infine, il numero di operazioni svolte per la ricostruzione

della sequenza di tagli è proporzionale al numero di alberi tagliati, dunque la complessità di questa ultima fase è pari a $O(N)$. In totale, quindi, l'intero algoritmo ha complessità $O(N^2)$.

■ Una soluzione $O(N)$

Prima di illustrare la soluzione lineare, introduciamo un altro paio di concetti importanti.

Definiamo *abbattitore di un albero* i, indicato con `abbattitore`$[i]$, l'indice del primo albero a destra di i, se esiste, che abbatte i quando cade verso sinistra; nel caso in cui l'abbattitore di i non esista, assegneremo convenzionalmente `abbattitore`$[i] = \infty$.

Definiamo inoltre *catena di abbattitori dell'albero* i la sequenza formata da i, dall'abbattitore di i, dall'abbattitore dell'abbattitore di i, e così via:

$$\text{catena di abbattitori di } i = i \to \texttt{abbattitore}[i] \to \texttt{abbattitore}[\texttt{abbattitore}[i]] \to \cdots,$$

dove l'iterazione viene troncata nel momento in cui giungiamo ad un albero che non ammette abbattitore. La catena di abbattitori di i non è mai vuota, perché contiene sempre almeno un elemento, cioè i stesso. Per fissare il concetto, consideriamo la figura 2.

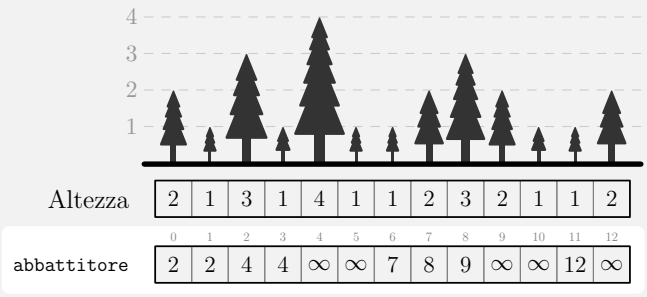


Figura 2

Nel caso in figura, ad esempio, l'abbattitore dell'albero 6 è l'albero 7, perché tra tutti gli alberi che lo abbattono se lasciati cadere a sinistra, 7 è il primo. La catena di abbattitori dell'albero 0 è $0 \to 2 \to 4$; la catena di abbattitori dell'albero 5 invece consiste del solo albero 5.

L'osservazione che permette di rendere la soluzione precedente più efficiente consiste nel notare che la catena di abbattitori dell'albero i è formata da tutti quegli alberi che sono in grado di abbattere i quando vengono lasciati cadere a sinistra. In altre parole, abbiamo appena constatato il fatto che gli alberi che abbattono i cadendo a sinistra sono disposti in maniera molto ordinata, e si raggiungono a partire dall'albero i passando di volta in volta da un albero al suo `abbattitore`.

Mostriamo ora come è possibile costruire velocemente le informazioni `lep`, `rep` e `abbattitore` per ogni albero:

```cpp
// Costruiamo rep in tempo lineare
for (int i = N - 1; i >= 0; i--) {
    rep[i] = i;
    while (rep[i] + 1 < N && rep[i] + 1 < i + H[i])
        rep[i] = rep[rep[i] + 1];
}
```

```cpp
// Costruiamo lep e abbattitore in tempo lineare
for (int i = 0; i < N; i++) {
    lep[i] = i, abbattitore[i] = INF;
    while (lep[i] - 1 >= 0 && lep[i] - 1 > i - H[i]) {
        abbattitore[lep[i] - 1] = i;
        lep[i] = lep[lep[i] - 1];
    }
}
```

L'idea alla base del calcolo è, a tutti gli effetti, quella di simulare l'*effetto domino*. Consideriamo ad esempio il calcolo di `rep` per l'albero i:

- inizialmente viene controllato se i, cadendo, abbatte l'albero $i + 1$;
- se sì, allora tutti gli alberi da $i + 1$ a `rep`$[i + 1]$ sono abbattuti dalla caduta di i, e `rep`$[i]$ viene temporaneamente impostato a `rep`$[i + 1]$;
- successivamente viene controllato se i, continuando la caduta, abbatte anche il primo albero non ancora caduto alla destra di i, cioè `rep`$[i + 1] + 1$;

- se sì, allora tutti gli alberi da `rep`$[i+1]+1$ a `rep`[`rep`$[i+1]+1$] sono abbattuti dalla caduta di i, e `rep`$[i]$ viene temporaneamente impostato a `rep`[`rep`$[i+1]+1$];
- ...

Dimostrare che questo metodo è in effetti capace di calcolare `lep`, `rep` e `abbattitore` di tutti gli alberi in tempo lineare è un semplice esercizio di analisi ammortizzata[1], ma non ce ne occuperemo qui per non appesantire la discussione.

Vediamo ora come è possibile velocizzare il calcolo di `risolvi`(i). L'implementazione che abbiamo visto prima era quadratica a causa del fatto che per ogni albero i è necessario considerare tutti gli alberi alla destra di i in grado di abbatterlo, una quantità di alberi ogni volta potenzialmente dell'ordine di N. Vedremo tra un attimo come evitare questa ricerca inutile, usando le informazioni sull'abbattitore di i.

Definiamo *migliore albero della catena di abbattitori di* i, d'ora in poi indicato con `migliore`$[i]$, l'albero j appartenente alla catena di abbattitori di i per cui è minima la quantità `risolvi`$(j+1)$.`numero_tagli`. Intuitivamente, `migliore`$[i]$ rappresenta, tra tutti gli alberi che sono in grado di abbattere i cadendo a sinistra, quello per cui risulta minimo il numero di tagli necessari per abbattere tutti gli alberi da i a $N-1$.

Ammettendo di essere in grado di calcolare velocemente `migliore`$[i]$ per ogni albero i, la funzione `risolvi` si ridurrebbe semplicemente a:

```
1  // risolvi(i) ritorna un oggetto info_t, che contiene
2  //   - numero_tagli: il minimo numero di tagli da effettuare per abbattere tutti gli alberi da
3  //                   i a N-1 inclusi.
4  //   - primo_albero: l'indice del primo albero da tagliare
5  //   - direzione:    la direzione della caduta del primo albero
6  info_t risolvi(int i) {
7      info_t risposta;
8
9      if (i == N) {
10         // Se non ci sono alberi da tagliare, numero_tagli = 0
11         risposta.numero_tagli = 0;
12     } else {
13         // Primo caso: abbatti i a destra
14         risposta.numero_tagli = risolvi(rep[i] + 1).numero_tagli + 1;
15         risposta.primo_albero = i;
16         risposta.direzione = DESTRA;
17
18         // Secondo caso: abbatti a sinistra migliore[i]
19         int j = migliore(i);
20         if (risolvi(j + 1).numero_tagli + 1 < risposta.numero_tagli) {
21             risposta.numero_tagli = risolvi(j + 1).numero_tagli + 1;
22             risposta.primo_albero = j;
23             risposta.direzione = SINISTRA;
24         }
25     }
26
27     return risposta;
28 }
```

dove la principale differenza con la versione quadratica consiste nell'aver eliminato il ciclo della riga 29.

Rimane solamente da capire come calcolare `migliore`$[i]$ in modo efficiente. Sicuramente, se `abbattitore`$[i] = \infty$, si avrà `migliore`$[i] = i$. Vice versa, supponiamo che esista l'abbattitore di i; per la natura della catena di abbattitori, per determinare il valore di `migliore`$[i]$ è sufficiente confrontare tra di loro gli alberi i e `migliore`[`abbattitore`$[i]$], e scegliere chi tra i due minimizza `risolvi`$(j+1)$.`numero_tagli`. Mostriamo una semplice implementazione in codice di questa idea all'inizio della prossima pagina.

[1] Per un'introduzione all'argomento dell'analisi ammortizzata si consideri ad esempio il documento disponibile al seguente indirizzo: http://goo.gl/7OegpB

```cpp
// migliore(i) ritorna l'albero j appartenente alla catena di abbattitori di i per cui
// è minima la quantità risolvi(j + 1).numero_tagli
int migliore(int i) {
    int risposta = i;

    // Se i ammette un abbattitore, confrontiamo l'albero i, col migliore albero della catena
    // di abbattitori dell'abbattitore di i
    if (abbattitore[i] != INF) {
        // j contiene il migliore albero della catena di abbattitori dell'abbattitore di i
        int j = migliore(abbattitore[i]);

        // Confrontiamo l'albero j con l'albero i
        if (risolvi(j + 1).numero_tagli < risolvi(i + 1).numero_tagli)
            risposta = j;
    }

    return risposta;
}
```

Ora che abbiamo tutti i tasselli del puzzle, non rimane che riutilizzare (inalterata) la funzione `ricostruisci_tagli` per risolvere completamente il problema.