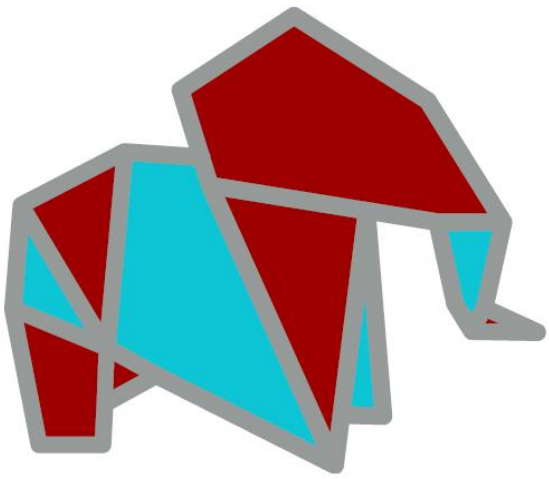
AICA
Associazione Italiana per l'Informatica
ed il Calcolo Automatico

Ministero dell'Istruzione
dell'Università e Ricerca



OII
CATANIA 2016

Catania, 15-17 Settembre 2016

FINALE NAZIONALE

Testi e soluzioni ufficiali dei problemi

Testi e soluzioni dei problemi

Giorgio Audrito, Alice Cortinovis, William Di Luigi, Gabriele Farina, Gaspare Ferraro,

Giada Franz, Roberto Grossi, Luigi Laura, Gemma Martini, Dario Ostuni, Romeo Rizzi

Coordinamento

Monica Gati

Supervisione a cura del Comitato per le Olimpiadi di Informatica

Classifica senza fili (`classifica`)

Limite di tempo:	0.8 secondi
Limite di memoria:	256 MiB
Difficoltà:	1

La pagina web che mostra la classifica in tempo reale delle OII è andata in crash, e i referenti regionali sono ora nel panico e non capiscono come stanno procedendo i loro beniamini! Per fortuna, loro sanno che prima del crash gli N concorrenti erano piazzati in un ordine ben preciso, per cui l'$(i+1)$-esimo in classifica era il concorrente `ids[`i`]`. Da quel momento, si possono basare soltanto sulle soffiate che gli arrivano dai tutor, impietositi dalla situazione. Più precisamente, i tutor fanno partire una nuova soffiata sulla gara in corso non appena:

- il concorrente `id` supera in classifica quello a lui immediatamente precedente;
- il concorrente `id` viene squalificato per aver tentato di hackerare il sistema di gara.[1]

I referenti, tuttavia, sono confusi da questa mole di dati perché vorrebbero semplicemente sapere chi si trova in alcune posizioni `pos` di loro interesse nella classifica. Aiutali tenendo traccia di tutte le soffiate, così da poter rispondere alle loro domande!

Implementazione

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

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

Dovrai implementare le seguenti funzioni:

C/C++	`void inizia(int N, int ids[]);`
Pascal	`procedure inizia(N: longint; ids: array of longint);`

- L'intero N rappresenta il numero di concorrenti.
- L'array `ids`, indicizzato da 0 a $N-1$, contiene i codici identificativi dei concorrenti (numeri a loro volta da 0 a $N-1$) nell'ordine in cui si trovavano prima del crash.

C/C++	`void supera(int id);`
Pascal	`procedure supera(id: longint);`

- L'intero `id` rappresenta il codice identificativo del concorrente che ha effettuato il sorpasso.

C/C++	`void squalifica(int id);`
Pascal	`procedure squalifica(id: longint);`

- L'intero `id` rappresenta il codice identificativo del concorrente che è stato squalificato.

[1] I tutor si accorgono *sempre* prontamente di ogni tale tentativo.

C/C++	`int partecipante(int pos);`
Pascal	`function partecipante(pos: longint): longint;`

- L'intero `pos` rappresenta la posizione nella classifica che un referente regionale vuole conoscere.
- La funzione dovrà restituire il codice identificativo del concorrente che al momento si trova in quella posizione nella classifica.

Il grader chiamerà prima la funzione `inizia`, poi le funzioni `supera`, `squalifica`, `partecipante` un qualunque numero di volte (e in un qualunque ordine) e stamperà i valori restituiti da `partecipante` sul file di output (nello stesso ordine in cui sono stati ottenuti).

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per verificare le vostre soluzioni in locale. Il grader di esempio legge i dati da `stdin`, chiama le funzioni che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da $Q + 2$ righe, dove Q è il numero totale di chiamate alle funzioni `supera`, `squalifica` e `partecipante`, contenenti:

- Riga 1: i due interi N e Q.
- Riga 2: i valori `ids[`i`]` per $i = 0 \dots N - 1$.
- Righe $3 \dots Q + 2$: la descrizione di una soffiata o richiesta, che può quindi essere:
  - `'s' id`: se `id` supera;
  - `'x' id`: se `id` viene squalificato;
  - `'p' pos`: se viene richiesta la posizione `pos`.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: i valori restituiti dalle chiamate alla funzione `partecipante` separati da spazio.

Assunzioni

- $1 \leq N \leq 1\,000\,000$.
- $1 \leq Q \leq 1\,000\,000$.
- $0 \leq$ `ids[`i`]` $\leq N - 1$ per ogni $i = 0 \dots N - 1$.
- `ids[`i`]` $\neq$ `ids[`j`]` per ogni $i \neq j$ (i numeri contenuti in `ids` sono tutti distinti).
- $0 \leq$ `id` $\leq N - 1$ nelle chiamate a `supera` e `squalifica`.
- Il primo in classifica non supera mai.
- Una volta che un partecipante viene squalificato, non può più superare né venire ulteriormente squalificato.
- La posizione `pos` nelle chiamate a `partecipante` esiste sempre.

Assegnazione del punteggio

Il tuo programma verrà verificato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo a un subtask, è necessario risolvere correttamente tutti i test che lo compongono.

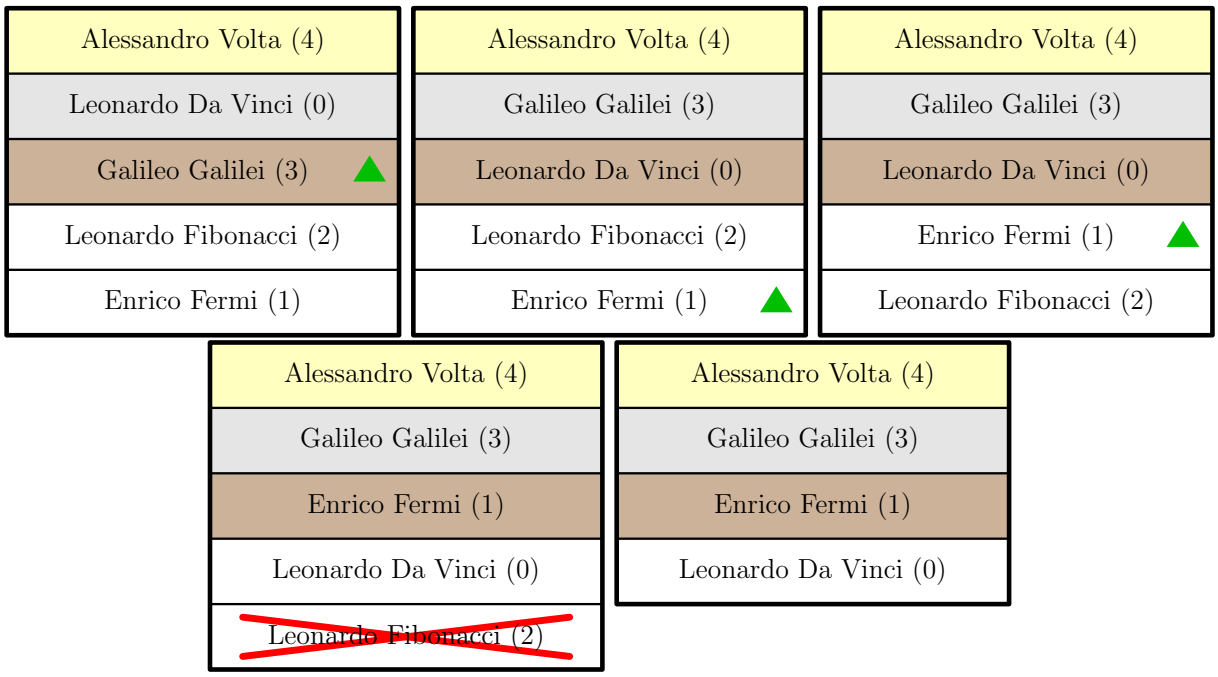
- **Subtask 1** [**0 punti**]: Casi d'esempio.
- **Subtask 2** [**18 punti**]: $N, Q \leq 10\,000$.
- **Subtask 3** [**16 punti**]: La funzione `squalifica` non viene mai chiamata. Inoltre $Q \leq 100\,000$.
- **Subtask 4** [**19 punti**]: La funzione `partecipante` viene chiamata solo dopo tutte le chiamate a `supera` e `squalifica`. Inoltre $Q \leq 100\,000$.
- **Subtask 5** [**17 punti**]: La funzione `supera` non viene mai chiamata.
- **Subtask 6** [**18 punti**]: $Q \leq 100\,000$.
- **Subtask 7** [**12 punti**]: Nessun limite specifico.

Esempi di input/output

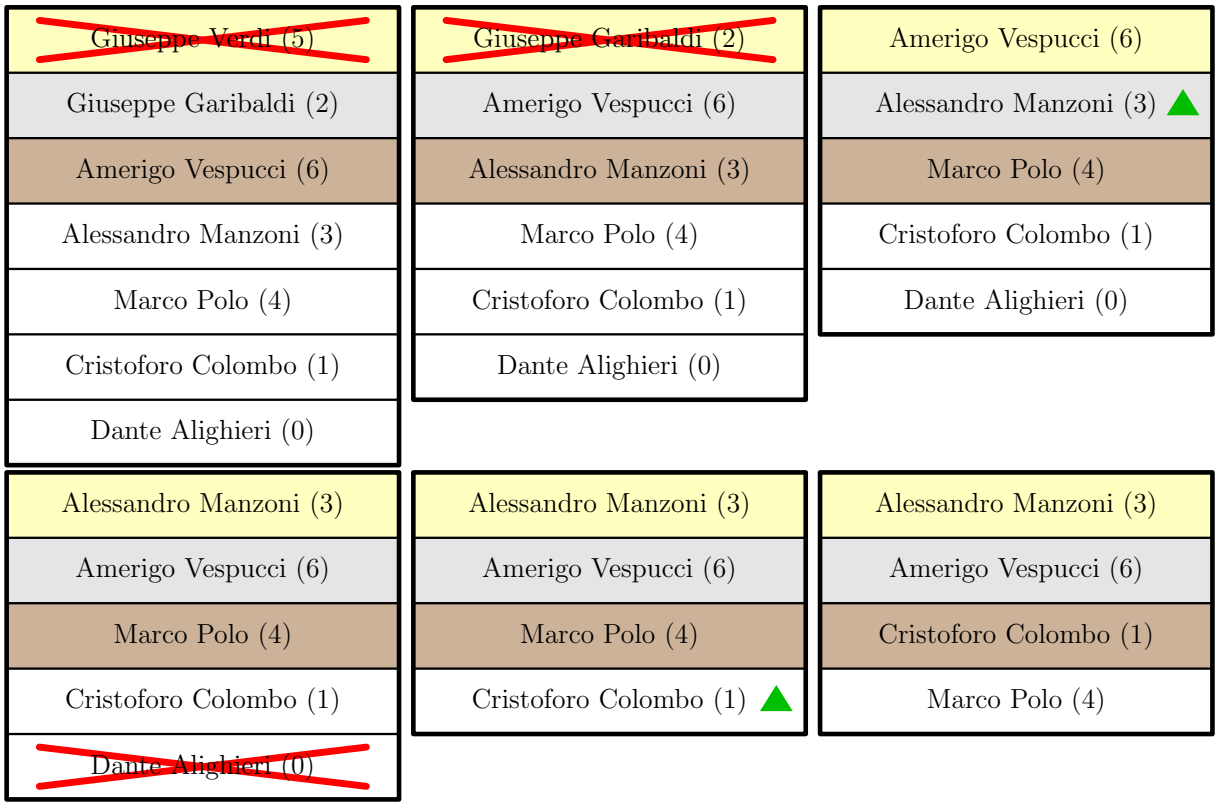
stdin	stdout
5 6 4 0 3 2 1 s 3 s 1 s 1 p 3 x 2 p 4	1 0
7 11 5 2 6 3 4 1 0 x 5 p 1 x 2 p 1 p 2 s 3 p 1 p 4 x 0 s 1 p 3	2 6 3 3 1 1

Spiegazione

Nel **primo caso di esempio**, la classifica si evolve come in figura:



Nel **secondo caso di esempio**, la classifica si evolve come in figura:



Soluzione

La soluzione che prende punteggio pieno ha complessità $O(\log N)$ per le funzioni `squalifica` e `partecipante` mentre ha complessità $O(1)$ per la funzione `supera`. Prima presentiamo due soluzioni subottimali: la prima simula nel modo più naturale possibile l'andamento della gara, mentre la seconda, usando una diversa rappresentazione della classifica, permette di eseguire le funzioni `supera` e `squalifica` in tempo costante e la funzione `partecipante` in tempo $O(N)$.

■ Simulare la classifica

Possiamo simulare l'andamento della gara utilizzando un array `classifica` contenente la classifica al momento del crash, cioè `classifica[`i`]` contiene l'`id` del concorrente in posizione $i + 1$.

Implementiamo la funzione `supera` cercando la posizione `pos` del concorrente `id` nell'array `classifica` e simuliamo il sorpasso scambiando i valori `classifica[pos]` e `classifica[pos-1]`. In questo modo la complessità di questa operazione è data dalla ricerca del concorrente in classifica ed è quindi lineare nel numero di concorrenti.

Per squalificare il concorrente `id` troviamo la sua posizione `pos` nell'array `classifica` e aggiorniamo la posizione dei concorrenti che stanno dietro di lui, shiftando perciò gli elementi di indice da `pos` $+ 1$ a $N - 1$ negli elementi di indice da `pos` a $N - 2$. Anche questa operazione risulta lineare nel numero di concorrenti.

A questo punto la funzione `partecipante` si può implementare in tempo costante, poiché l'id del concorrente in posizione `pos` è contenuto dentro a `classifica[pos-1]`.

■ Classifica salvata tramite una lista

È possibile simulare la classifica salvandosi per ogni partecipante, se è ancora in gara, l'id di quello immediatamente prima e l'id di quello immediatamente dopo di lui in classifica. Teniamo queste informazioni in due array `precedente` e `successivo`, dove convenzionalmente il primo partecipante in classifica ha come precedente -1 e l'ultimo ha come successivo N.

Entrambe le operazioni `supera` e `squalifica` possono essere implementate in tempo costante tramite un opportuno aggiornamento degli array `precedente` e `successivo`. Per esempio, per quanto riguarda la funzione squalifica, l'aggiornamento da compiere sarà `successivo[precedente[id]] = successivo[id]` e `precedente[successivo[id]] = precedente[id]`.

Per quanto riguarda la funzione `partecipante`, è sufficiente partire dal concorrente che sta primo in classifica (cioè quello tale che `precedente[id]` $= -1$) e passare al suo successivo, poi al successivo di questo, e così via fino ad arrivare alla posizione desiderata. In questo modo, la complessità di ogni chiamata alla funzione è lineare nel numero dei partecipanti.

■ Soluzione ottima

Presentiamo infine la soluzione ottimale del problema, che utilizza un *range tree* per implementare velocemente tutte le funzioni richieste. In particolare saremo in grado di implementare il sorpasso in $O(1)$ e la squalifica e la ricerca di una posizione in $O(\log N)$.

Un punto a sfavore della prima soluzione presentata è che quando si squalifica un concorrente, bisogna aggiornare tutto l'array `classifica`. Invece di eliminare effettivamente i concorrenti dall'array `classifica`, teniamo un array di booleani `pos_valide`, in cui `pos_valide[`i`]`$=$ `false` se il concorrente

in `classifica[`i`]` è stato squalificato. In questo modo un concorrente in `classifica[`i`]` non è più necessariamente l'$(i+1)$-esimo in classifica: quindi serve un modo per ricostruire velocemente la classifica reale.

In particolare costruiamo un albero binario sopra l'array `classifica` che in ogni nodo salva il numero di posizioni ancora valide nel sottoalbero corrispondente. In questo modo navigando l'albero, che avrà altezza $\log N$, è possibile individuare il partecipante nell'i-esima posizione della classifica in tempo logaritmico.

Parallelamente all'albero binario teniamo due array `precedente` e `successivo` come nella soluzione precedente, per poter attuare il sorpasso in $O(1)$, e un array `id2pos` che memorizza in che indice dell'array `classifica` si trova ogni partecipante, in modo da individuare in tempo costante le posizioni su cui agire con `supera` e `squalifica`.

In seguito trovate un'immagine che esemplifica l'implementazione del seguente caso di input:

stdin	stdout
7 3 5 2 6 3 4 1 0 x 5 p 2 s 0	6

In figura 1, si inizializza il range tree ponendo in ogni foglia l'id del partecipante in quella posizione e in ogni nodo il numero di partecipanti in gara sotto di lui.

Per implementare la funzione `squalifica(5)`, come si può vedere in figura 2, si cancella l'id 5 (in rosso in figura) e si aggiornano tutti i valori dei nodi suoi antenati (in arancione).

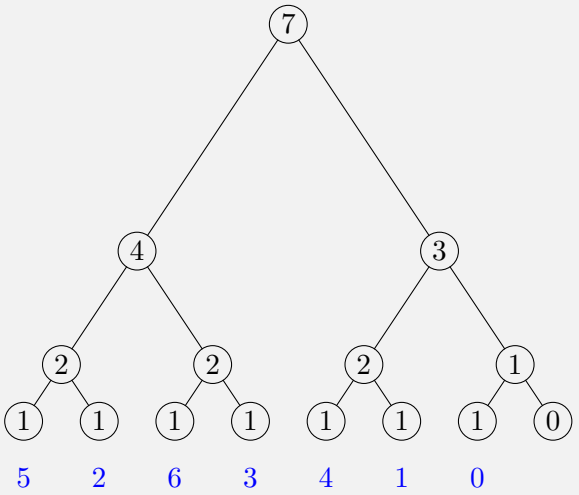


Figura 1: `inizia`

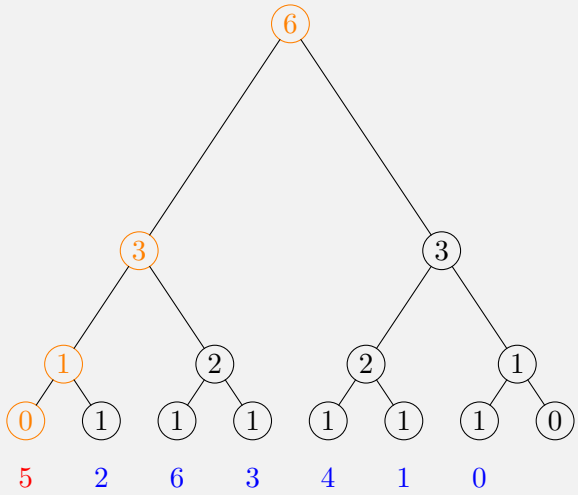


Figura 2: `squalifica(5)`

Per conoscere invece chi sta in posizione 2, si scende nell'albero attraversando i nodi arancioni in figura 3, decidendo dove dirigersi in base a quanti partecipanti attivi ci sono in ogni nodo. Si giunge alla foglia che ha l'id 6 e che è effettivamente il secondo in classifica.

Infine, si implementa il sorpasso del partecipante con id 0 scambiando gli id 0 e 1 nelle due foglie corrispondenti (arancioni in figura 4).

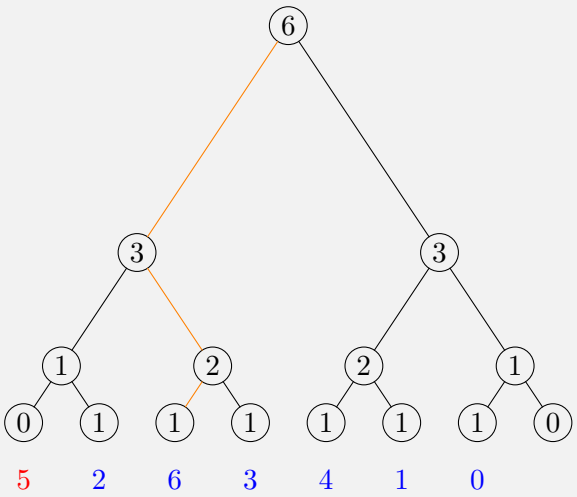


Figura 3: `partecipante(2)`

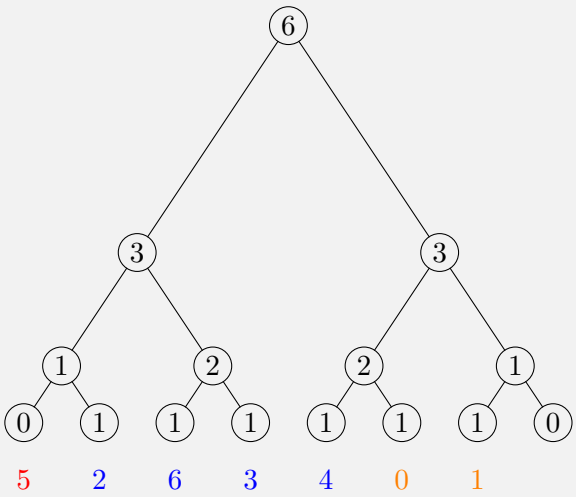


Figura 4: `supera(0)`

Esempio di codice C++11

```cpp
const int MAXID = 1000010, MAXN = 1<<20;

int N, rt[2*MAXN]; // range-tree
int id2pos[MAXID], classifica[MAXN], successivo[MAXN], precedente[MAXN];

// Aggiorno il range tree squalificando il concorrente pos-esimo
void azzera(int pos) {
        int it = MAXN+pos;
        while (it != 0) {
                rt[it]--;
                it >>= 1;
        }
}
// Trova l'indice del pos-esimo 1
int pos2rank(int pos) {
        int it = 1;
        while (it < MAXN) {
                if (rt[it<<1] < pos) {
                        pos -= rt[it<<1];
                        it = (it<<1)|1;
                }
                else it = it<<1;
        }
        return it-MAXN;
}
void inizia(int N, int* ids) {
        ::N = N;
    // Inizializzo gli array per ogni concorrente inizialmente in gara
        for (int i = 0; i < N; i++) {
                id2pos[ids[i]] = i;
                classifica[i] = ids[i];
                successivo[i] = i-1;
                precedente[i] = i+1;
                rt[MAXN+i] = 1;
        }
    // Inizializzo range tree
        for (int i = MAXN-1; i >= 1; i--) rt[i] = rt[i<<1] + rt[(i<<1)|1];
}
void squalifica(int id) {
    // Ritrovo gli indici in classifica del precedente e successivo
        // del concorrente che vogliamo eliminare
        int pos = id2pos[id];
        int successivo = successivo[rank];
        int precedente = precedente[rank];
    // Se esistono, aggiorno precedente e successivo
        if (successivo >= 0) precedente[successivo] = precedente;
        if (precedente < N) successivo[precedente] = successivo;
        azzera(pos); // Aggiorno la bitmask
}
// Trovo gli indici corretti del concorrente e del suo successivo in classifica
// infine scambio i valori in classifica e posizione
void supera(int id1) {
        int pos1 = id2pos[id1];
        int pos2 = successivo[pos1];
        int id2 = classifica[pos2];
        std::swap(id2pos[id1], id2pos[id2]);
        std::swap(classifica[pos1], classifica[pos2]);
}
// Ritrovo l'indice del pos-esimo "1" nella bitmask e stampo il concorrente richiesto
int partecipante(int pos) {
        int rank = pos2rank(pos);
        return classifica[rank];
}
```

Scale di Hogwarts (`hogwarts`)

Limite di tempo:	1.0 secondi
Limite di memoria:	256 MiB
Difficoltà:	1

A Hogwarts, la più prestigiosa scuola di magia del mondo, si sa che *alle scale piace cambiare*! Dopo un lungo viaggio in treno e la cerimonia di smistamento, sei pronto per la tua prima lezione: Pozioni. Il castello ha N sale, numerate da 0 a $N-1$, tra loro collegate da M scale. Il tuo dormitorio si trova nella sala 0 e la lezione di Pozioni si tiene nell'aula allestita nella stanza $N-1$, nei sotterranei del castello. Fortunatamente, prima di intraprendere il percorso attraverso le sale, conosci l'orario in cui compare e quello in cui scompare ogni scala. Per percorrere una scala impieghi esattamente 1 minuto, ma puoi sostare nelle sale per tutto il tempo che ritieni necessario.

Il professore di Pozioni è molto severo e non tollera ritardi, dunque devi assolutamente arrivare a lezione nel minor tempo possibile. Trova un modo per raggiungere l'aula nel minimo tempo possibile, se un modo per raggiungerla esiste!

Implementazione

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

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

Dovrai implementare la seguente funzione:

C/C++	`int raggiungi(int N, int M, int A[], int B[], int inizio[], int fine[]);`
Pascal	`function raggiungi(N,M: longint; A,B,inizio,fine: array of longint): longint;`

- L'intero N rappresenta il numero di sale del castello.
- L'intero M rappresenta il numero di scale.
- Gli array `A`, `B`, `inizio` e `fine` sono indicizzati da 0 a $M-1$ e contengono le informazioni sulla comparsa e sparizione delle scale: l'i-esima scala collega tra loro le sale `A[`i`]` e `B[`i`]`, compare al tempo `inizio[`i`]` e scompare al tempo `fine[`i`]`.
- La funzione deve restituire il minimo tempo necessario per andare dalla sala 0 alla sala $N-1$; se non è possibile raggiungere la sala $N-1$, deve restituire il valore -1.

Il grader chiamerà la funzione `raggiungi` e ne stamperà il valore restituito sul file di output.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per verificare le vostre soluzioni in locale. Il grader di esempio legge i dati da `stdin`, chiama la funzione che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da $M+1$ righe, contenenti:

- Riga 1: gli interi N e M.
- Righe $2, \ldots, M+1$: l'i-esima di queste righe contiene, nell'ordine, i valori `A[`i`]`, `B[`i`]`, `inizio[`i`]` e `fine[`i`]` per $i = 0, \ldots, M-1$.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: il valore restituito dalla funzione `raggiungi`.

Assunzioni

- $2 \leq N \leq 500\,000$.
- $1 \leq M \leq 1\,000\,000$.
- $0 \leq \texttt{A}[i], \texttt{B}[i] \leq N - 1$ per ogni $i = 0, \ldots, M - 1$.
- Non ci sono scale che collegano una sala a se stessa ($\texttt{A}[i] \neq \texttt{B}[i]$).
- Non ci sono due o più scale che collegano le stesse due sale.
- Ogni scala è percorribile in una qualunque delle due direzioni.
- $0 \leq \texttt{inizio}[i] < \texttt{fine}[i] \leq 2\,000\,000$.

Assegnazione del punteggio

Il tuo programma verrà verificato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo a un subtask, è necessario risolvere correttamente tutti i test che lo compongono.

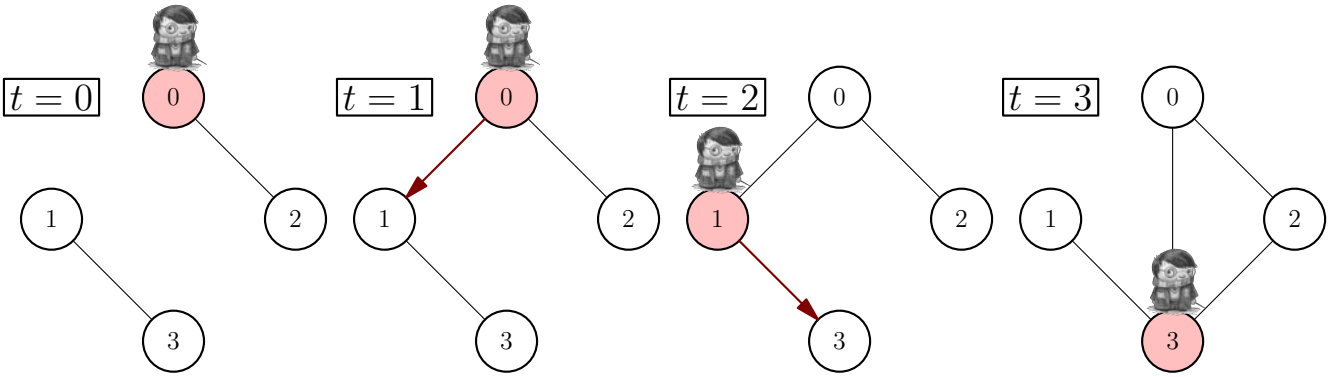
- **Subtask 1 [0 punti]**: Casi d'esempio.
- **Subtask 2 [10 punti]**: $N \leq 10, M \leq 15$ e $\texttt{fine}[i] \leq 20$ per ogni i.
- **Subtask 3 [21 punti]**: Tutte le scale sono fisse, cioè il tempo di inizio è 0 per tutte e il tempo di fine è uguale per tutte.
- **Subtask 4 [18 punti]**: Le scale scompaiono soltanto, cioè il tempo di inizio è 0 per tutte.
- **Subtask 5 [22 punti]**: $N \leq 1000$, $M \leq 2000$, $\texttt{fine}[i] \leq 5000$ per ogni i.
- **Subtask 6 [29 punti]**: Nessuna limitazione specifica.

Esempi di input/output

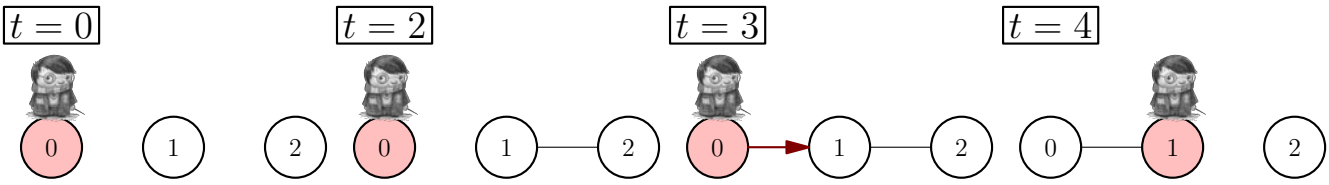
stdin	stdout
4 5 0 2 0 5 0 1 1 3 0 3 3 6 3 2 3 8 3 1 0 10	3
3 2 0 1 3 5 1 2 2 4	-1

Spiegazione

Nel **primo caso di esempio** il modo più veloce per andare dalla sala 0 alla 3 è aspettare 1 minuto, poi prendere la scala che collega 0 e 1 (ci metti 1 minuto) e poi prendere immediatamente la scala che collega 1 e 3 (anche qui ci metti 1 minuto), impiegando in totale 3 minuti per arrivare a lezione.



Nel **secondo caso di esempio** non è possibile andare dalla sala 0 alla sala 2! Infatti dovresti necessariamente passare per la sala 1 perché non c'è mai una scala che collega direttamente 0 e 2. Al tempo 3 compare una scala per andare da 0 a 1, dunque puoi trovarti nella stanza 1 al più presto dopo 4 minuti, e in quell'istante scompare la scala che collega le sale 1 e 2, impedendoti di raggiungere la destinazione.



Soluzione

Presentiamo due soluzioni complete del problema, ma prima vediamo le soluzioni di alcuni subtask che contengono idee utili per le soluzioni ottimali. Nel corso di questa nota, chiamiamo $T_{\max}$ il tempo di scomparsa dell'ultima scala.

■ Se le scale sono fisse

Se tutte le scale sono fisse, cioè compaiono tutte al tempo 0 e scompaiono tutte al tempo $T_{\max}$, il problema si risolve con una BFS (Breadth First Search o ricerca in ampiezza) sul grafo i cui vertici sono le sale e i cui archi sono le scale. Il *peso* di un arco è il tempo impiegato per percorrere la scala corrispondente, dunque tutti gli archi hanno peso 1. Quindi tramite una BFS è possibile sapere se è possibile andare dal vertice 0 al vertice $N-1$ e, nel caso sia possibile, il minimo tempo t necessario per farlo. L'unico accorgimento è che se $t > T_{\max}$ le scale scompaiono prima che si riesca ad arrivare alla stanza $N-1$ e quindi la risposta dovrà essere -1. Questa è la soluzione del subtask 3 e ha complessità computazionale lineare nel numero di archi.

■ Se le scale possono soltanto scomparire

Se le scale sono tutte presenti all'inizio, ma possono scomparire, si può risolvere il problema modificando la BFS: quando ci troviamo in un nodo x a distanza d dal vertice 0 e vogliamo percorrere un arco che va da x a y, lo possiamo percorrere solo se d è strettamente minore del momento in cui scompare l'arco in questione. Anche questa soluzione ha complessità computazionale lineare nel numero di archi.

■ Soluzione $O(M \log M)$

Presentiamo ora una prima soluzione del problema con archi che possono sia comparire che scomparire. Il punto di partenza è l'algoritmo di Dijkstra per trovare il cammino di lunghezza minima su un grafo pesato con pesi non negativi. È sufficiente modificare l'algoritmo nel seguente modo: quando ci si "espande" da un nodo x che sta a distanza t dal vertice 0 percorrendo un arco (x, y) che compare al tempo T_{inizio} e scompare al tempo T_{fine}, si controlla innanzitutto se T_{fine} è minore o uguale a d (cioè se l'arco è già scomparso). In caso di risposta negativa, bisogna verificare se il collegamento con y è già presente al tempo t oppure no, dunque il tempo in cui si può raggiungere y sarà 1 più il massimo tra T_{inizio} e t. Questa soluzione ha complessità $O(M \log M)$.

■ Soluzione $O(M + T_{\max})$

Presentiamo ora la seconda soluzione del problema. L'idea è di trovare, per ogni tempo t, quali sono le sale che possono essere raggiunte *esattamente* al tempo t (e non prima). Al tempo $t = 0$, l'unica sala che può essere raggiunta è la sala 0. Teniamo una variabile `vector <int> raggiunti[MAXT+1]` in cui in `raggiunti[`t`]` metteremo i nodi che sono raggiungibili esattamente al tempo t, più alcuni nodi raggiungibili prima di t che vedremo come trattare in seguito.

Quando siamo al tempo t, vogliamo sapere quali nuovi vertici possiamo raggiungere a partire dai vertici in `raggiunti[`t`]` (che potranno essere raggiunti dal tempo $t+1$ in poi, a seconda di quando compaiono gli archi uscenti dai vertici di `raggiunti[`t`]`). Per farlo, consideriamo uno alla volta i vertici in `raggiunti[`t`]`. Sia x un tale vertice. Per ogni arco (x, y) che compare al tempo T_{inizio} e scompare al tempo T_{fine} possono verificarsi quattro situazioni differenti:

1. la scala (x, y) è inutile perché anche y è raggiungibile entro il tempo t; in questo caso semplicemente ignoriamo l'arco;
2. la scala (x, y) non è più utilizzabile perché l'arco è scomparso prima che x fosse raggiungibile, cioè $T_{\text{fine}} \leq t$; anche in questo caso ignoriamo l'arco;
3. la scala (x, y) è utile e percorribile immediatamente, allora dobbiamo segnare che y è raggiungibile *entro* il tempo $t + 1$ (perché comunque ci mettiamo 1 secondo per percorrere la scala (x, y), aggiungendo il nodo y a `raggiunti[`$t + 1$`]`);
4. la scala (x, y) è utile ma *al momento* non è utilizzabile immediatamente perché compare dopo il tempo t; in questo caso, detto T_{inizio} il momento in cui compare la scala, segneremo che y è raggiungibile *entro* il tempo $T_{\text{inizio}} + 1$ aggiungendo il nodo y a `raggiunti[`$T_{\text{inizio}} + 1$`]`.

Può accadere di segnare un nodo x in più di un elemento dell'array di vector `raggiunti`; è quindi indispensabile tenere una variabile `bool fatto[MAXN+1]` in cui nella casella x viene messo `true` quando un nodo è raggiungibile per la prima volta (cioè la prima volta che, scorrendo i tempi t, incontriamo il nodo nella variabile `raggiunti[`t`]`). In questo modo, se incontriamo di nuovo il nodo x sappiamo che dobbiamo ignorarlo.

Poiché guardiamo ogni arco al più una volta, e analizziamo ogni tempo t da 0 a $T_{\max}$, la complessità computazionale della soluzione proposta è $O(M + T_{\max})$.

Esempio di codice C++11

Di seguito trovate un'implementazione della soluzione ottimale di questo problema.

```cpp
#include <vector>

const int INF = 1000000000;
const int MAXT = 5000000;
const int MAXN = 1000000;

std::vector < std::pair <int, int> > archi[MAXN+1];

std::vector <int> raggiunti[MAXT+1];
bool fatto[MAXN+1];
int dist[MAXN+1];

int raggiungi(int N, int M, int A[], int B[], int inizio[], int fine[]) {
    // Creazione del grafo
    for (int i = 0; i < M; i++) {
        archi[A[i]].push_back(make_pair(i, B[i]));
        archi[B[i]].push_back(make_pair(i, A[i]));
    }
    // Inizializzazione
    for (int i = 0; i < N; i++) {
        fatto[i] = false;
        dist[i] = INF;
    }

    // Il nodo 0 è l'unico raggiungibile al tempo 0
    raggiunti[0].push_back(0);
    dist[0] = 0;

    for (int t = 0; t <= MAXT; t++) {
        for (int v : raggiunti[t]) {
            // Processo i vertici in raggiunti[t] che non sono
            // stati raggiunti prima di t
            if (!fatto[v]) {
                for (const auto& arco : archi[v]) {
                    int scala = arco.first;
                    int amico = arco.second;
                    int tempo = max(dist[v], inizio[scala])+1;
                    if (!fatto[amico] & dist[v] < fine[scala] & tempo < dist[amico]){
                        dist[amico] = tempo;
                        raggiunti[tempo].push_back(amico);
                    }
                }
                fatto[v] = true;
            }
        }
    }
    return (dist[N-1] == INF)? -1 : dist[N-1];
}
```

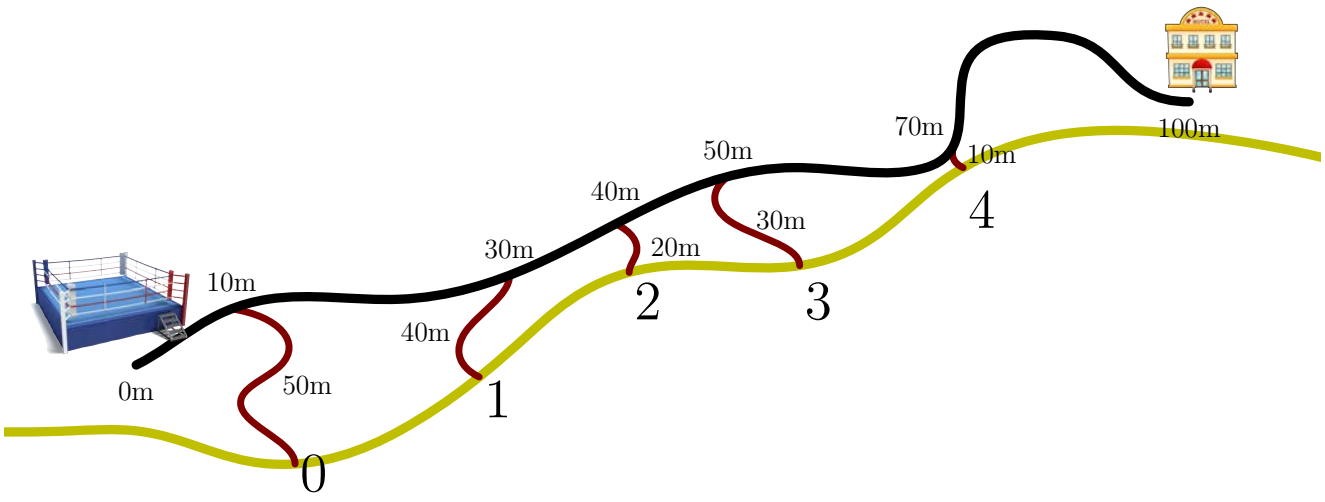
Solleone sul lungomare (`lungomare`)

Limite di tempo:	0.3 secondi
Limite di memoria:	512 MiB
Difficoltà:	1

Mojito, il Jackrussel di Monica, si è recato come tutti gli anni alla Gara Nazionale delle Olimpiadi Italiane di Informatica per svolgere la sua delicata mansione di mascotte. Ora che ha terminato il suo compito nella sede della gara, vuole tornare all'hotel dove una comoda cuccia lo aspetta! Per tornare all'hotel non deve fare altro che seguire un tratto di lungomare lungo L metri, senza alcuna deviazione. Ma l'obiettivo apparentemente semplice è reso complicato dal proverbiale solleone di Catania!

Infatti, Mojito rifugge la calura e vuole evitare di restare esposto al sole troppo a lungo. Per fortuna lungo il tragitto ci sono N accessi ad altrettante spiagge, l'i-esimo dei quali a una distanza D_i dall'inizio del percorso e collegato al bagnasciuga da una passerella di lunghezza P_i. A Mojito non importa quanto a lungo dovrà camminare in totale, ma solo di non restare sotto il sole per troppo tempo di seguito: come è noto, facendo un breve bagno in mare tutta la calura accumulata scompare!

Per esempio, consideriamo il seguente possibile lungomare:



In questo caso, se Mojito percorre tutto il lungomare senza fare bagni, camminerà per $C = L = 100$ metri continuamente sotto il sole. Se invece si ferma a fare il bagno in tutte le spiagge, il percorso assolato più lungo ce lo avrà tra le spiagge 0 e 1 e sarà pari a $C = 50 + 20 + 40 = 110$ metri. La cosa migliore è invece fermarsi nelle spiagge 2 e 4, nel qual caso le tre tappe saranno lunghe rispettivamente 60, 60 e 40 metri, e quindi $C = 60$.

Aiuta Mojito calcolando C, il minimo numero per cui è possibile andare dall'inizio alla fine del lungomare senza mai stare per più di C metri sotto il sole senza fare un bagno!

Implementazione

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

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

Dovrai implementare la seguente funzione:

C/C++	`long long percorri(int N, long long L, long long D[], long long P[]);`
Pascal	`function percorri(N: longint; L: int64; D, P: array of int64): int64;`

- L'intero N rappresenta il numero di spiagge presenti.
- L'intero L rappresenta la lunghezza in metri del lungomare.
- L'array `D`, indicizzato da 0 a $N-1$, contiene le distanze degli ingressi delle spiagge dall'inizio del lungomare.
- L'array `P`, indicizzato da 0 a $N-1$, contiene le lunghezze delle passerelle.

Il grader chiamerà la funzione `percorri` e ne stamperà il valore restituito sul file di output.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per verificare le vostre soluzioni in locale. Il grader di esempio legge i dati da `stdin`, chiama le funzioni che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da tre righe, contenenti:

- Riga 1: i due interi N e L.
- Riga 2: i valori `D[`i`]` per $i = 0 \dots N-1$.
- Riga 2: i valori `P[`i`]` per $i = 0 \dots N-1$.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: il valore restituito dalla funzione `percorri`.

Assunzioni

- $1 \leq N \leq 10\,000\,000$.
- $1 \leq L \leq 10^{18}$.
- $0 <$`D[`i`]`$<$`D[`$i+1$`]`$< L$ per ogni $i = 0 \dots N-1$.
- `P[`i`]` $\leq 10^{18}$ per ogni $i = 0 \dots N-1$.

Assegnazione del punteggio

Il tuo programma verrà verificato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo a un subtask, è necessario risolvere correttamente tutti i test che lo compongono.

- **Subtask 1** [**0 punti**]: Casi d'esempio.
- **Subtask 2** [**13 punti**]: Le passerelle `P[`i`]` sono tutte lunghe uguali e la distanza tra ogni due spiagge consecutive `D[`$i+1$`]` − `D[`i`]` è costante.
- **Subtask 3** [**11 punti**]: Le passerelle `P[`i`]` sono tutte lunghe uguali e $N \leq 100$.
- **Subtask 4** [**12 punti**]: Le passerelle `P[`i`]` sono tutte lunghe uguali e $N \leq 10\,000$.
- **Subtask 5** [**20 punti**]: $N \leq 15$.
- **Subtask 6** [**10 punti**]: $N \leq 10\,000$.
- **Subtask 7** [**23 punti**]: $N \leq 1\,000\,000$.
- **Subtask 8** [**11 punti**]: Nessuna limitazione specifica.

Esempi di input/output

stdin	stdout
5 100 10 30 40 50 70 50 40 20 30 10	60
10 200 5 20 50 70 95 100 125 150 155 160 10 20 15 25 12 20 25 15 30 30	67

Spiegazione

Il **primo caso di esempio** è quello descritto nel testo.

Nel **secondo caso di esempio** conviene fermarsi nelle spiagge 2, 3, 4, 6 e 7 (ma è possibile fermarsi anche nella spiaggia 1 ottenendo il medesimo risultato).

Soluzione

Osserviamo innazitutto che la distanza fra due spiagge di indici $u < v$ è $\text{dist}(u, v) \coloneqq \texttt{P[}u\texttt{]} - \texttt{D[}u\texttt{]} + \texttt{P[}v\texttt{]} + \texttt{D[}v\texttt{]}$. Scelti quindi le spiagge $u_1 < u_2 < \ldots < u_k$, il numero di metri che Mojito deve percorrere consecutivamente sotto il sole è il massimo fra $\texttt{P[}u_1\texttt{]} + \texttt{D[}u_1\texttt{]}$, $\text{dist}(u_i, u_{i+1})$ per $i = 1, \ldots, k-1$ e $\texttt{P[}u_k\texttt{]} + L - \texttt{D[}u_k\texttt{]}$.

Per evitare l'asimmetria del primo e dell'ultimo termine, possiamo immaginare di aggiungere due spiagge fittizie all'inizio e alla fine del lungomare con lunghezza nulla della passerella e risolvere lo stesso problema in cui però bisogna partire dalla prima spiaggia e arrivare all'ultima spiaggia. Infatti, in questo modo, scelte $u_1 < u_2 < \ldots < u_k$ spiagge in cui fermarsi (in cui u_1 è il lido fittizio all'inizio del lungomare e u_k è quello alla fine), il numero di metri che Mojito deve percorrere consecutivamente sotto il sole è $\text{calura}(u_1, \ldots, u_k) \coloneqq \max\{\text{dist}(u_i, u_{i+1}) \ : \ i = 1, \ldots, k\}$.

Presentiamo prima le idee chiave delle soluzioni di alcuni subtask, e infine la soluzione ottimale.

■ Se le passerelle sono tutte lunghe uguali

Se sia la lunghezza delle passerelle che la distanza tra spiagge consecutive sono costanti, si può facilmente dimostrare che la strategia ottimale è una di queste tre:

- non fermarsi in nessuna spiaggia, per cui la calura finale è $C = L$;
- fermarsi in tutte le spiagge;
- fermarsi esattamente in una spiaggia (quella in posizione più centrale).

Si può verificare semplicemente il risultato di tutte e tre queste strategie simulandole in tempo $O(N)$; o con qualche accorgimento matematico anche in tempo $O(1)$.

Se la lunghezza delle passerelle è costante, si può dimostrare che la strategia ottimale consiste nel fermarsi in tutto un intervallo contiguo di spiagge. Più precisamente, date la prima e l'ultima spiaggia in cui ci si ferma u_2 ed u_{k-1}, le spiagge intermedie in cui ci si deve fermare saranno esattamente $u_2 + 1, u_2 + 2, \ldots, u_{k-1} - 1, u_{k-1}$. È quindi possibile cercare una soluzione calcolando la calura di tutti gli N^2 intervalli di spiagge (in tempo $O(N)$ ciascuno) e scegliendo quello ottimale in tempo $O(N^3)$.

Se poi si fissa l'estremo iniziale, si possono provare tutti i possibili estremi finali corrispondenti in tempo $O(N)$ aggiornando incrementalmente la risposta, e quindi scoprendo la strategia ottimale in tempo $O(N^2)$.

■ Soluzione con complessità $O(N \log L)$

Presentiamo una prima soluzione (non ottimale) del problema. Data una lunghezza S, riusciamo a sapere in tempo lineare nel numero di spiagge se è possibile scegliere delle spiagge in modo che Mojito non percorra mai più di S metri consecutivi sotto al sole. Illustriamo ora l'algoritmo per rispondere alla domanda.

Quando Mojito arriva alla passerella i-esima avendo percorso m metri sotto al sole dopo l'ultima spiaggia in cui ha fatto il bagno, si ferma alla spiaggia i-esima se e solo se valgono le due condizioni seguenti:

- può fermarsi, cioè $m + \texttt{P[}i\texttt{]} \leq S$;
- gli conviene fermarsi, cioè $\texttt{P[}i\texttt{]} < m$.

Dopo l'i-esima spiaggia, se Mojito non si è fermato a m verrà aggiunto $\texttt{D}[i+1] - \texttt{D}[i]$, se Mojito si è fermato invece il valore di m verrà impostato a $\texttt{P}[i] + \texttt{D}[i+1] - \texttt{D}[i]$. Se ad un certo punto il valore di m supera S vuol dire che non è possibile percorrere meno di S metri consecutivi sotto al sole.

Per trovare la risposta, dobbiamo trovare il minimo S tale che si riesca a non percorrere mai più di S metri consecutivi sotto al sole. Questo può essere fatto con una ricerca binaria sui numeri da 1 a L (infatti la risposta sarà sempre inferiore o uguale a L), per cui la complessità computazionale di questo algoritmo è $O(N \log L)$.

■ Soluzione con complessità $O(N)$

Ricordiamo che la distanza fra due spiagge di indici $u < v$ è $\text{dist}(u, v) = \texttt{P}[u] - \texttt{D}[u] + \texttt{P}[v] + \texttt{D}[v]$. Questo ci fa pensare che data una spiaggia u, le due quantità interessanti siano $\texttt{P}[u] - \texttt{D}[u]$ e $\texttt{P}[u] + \texttt{D}[u]$, la prima interessante quando si esce dalla spiaggia e la seconda quando ci si entra.

In particolare supponiamo di essere fermi ad una spiaggia u e di voler scegliere il prossimo nodo v in cui fermarsi. Se $\texttt{P}[v] - \texttt{D}[v] \geq \texttt{P}[u] - \texttt{D}[u]$, possiamo evitare di fermarci nella spiaggia v, poiché dato un generico nodo $w > v > u$ abbiamo che $\text{dist}(u, w) \leq \text{dist}(v, w)$ e quindi ci conviene direttamente andare da u a w senza passare per v.

Inoltre vorremmo che se ci fermiamo in una spiaggia v allora $\texttt{P}[v] + \texttt{D}[v]$ sia "piccolo", poiché pagheremo questa quantità quando ci entreremo.

Cerchiamo ora di formalizzare quanto detto finora attraverso i seguenti lemmi.

Lemma. *Siano $u_1, \ldots, u_k$ le spiagge in cui Mojito si ferma in una soluzione ottima (per quanto detto all'inizio supponiamo che u_1 sia la spiaggia fittizia iniziale e u_k quella finale), allora possiamo supporre senza perdità di generalità che $\texttt{P}[u_{i+1}] - \texttt{D}[u_{i+1}] < \texttt{P}[u_i] - \texttt{D}[u_i]$ e $\texttt{P}[u_{i+1}] + \texttt{D}[u_{i+1}] > \texttt{P}[u_i] + \texttt{D}[u_i]$.*

Dimostrazione. Supponiamo che esista $1 \leq h < k$ tale che $\texttt{P}[u_{h+1}] - \texttt{D}[u_{h+1}] \geq \texttt{P}[u_h] - \texttt{D}[u_h]$, allora osserviamo che

$$\text{calura}(u_1, \ldots, u_h, u_{h+2}, \ldots, u_k) \leq \text{calura}(u_1, \ldots, u_k) \ .$$

Infatti, ricordando la definizione di calura, l'unico termine che compare nel massimo che definisce $\text{calura}(u_1, \ldots, u_h, u_{h+2}, \ldots, u_k)$ e non compare il quello che definisce $\text{calura}(u_1, \ldots, u_k)$ è $\text{dist}(u_h, u_{h+2})$, che però è minore o uguale a $\text{dist}(u_{h+1}, u_{h+2})$ che compare invece nella definizione di $\text{calura}(u_1, \ldots, u_k)$.

Perciò se $u_1, \ldots, u_k$ era una scelta di spiagge che dava una soluzione ottima, allora lo è anche $u_1, \ldots, u_h, u_{h+2}, \ldots, u_k$ e così abbiamo dimostrato la prima assunzione, ma in modo analogo si dimostra anche l'altra disuguaglianza. □

Lemma. *Siano $u_1, \ldots, u_k$ le spiagge in cui Mojito si ferma in una soluzione ottima, allora possiamo supporre che per ogni $1 \leq i < k$ valga che u_{i+1} è la spiaggia che minimizza $\texttt{P} - \texttt{D}$ fra le spiagge v successive a u per cui vale $\texttt{P}[v] - \texttt{D}[v] < \texttt{P}[u_i] - \texttt{D}[u_i]$.*

Dimostrazione. Supponiamo che la tesi non sia rispettata dalle spiagge $u_1, \ldots, u_k$ per l'indice $1 \leq h < k$ e sia $v > u$ la spiaggia che minimizza $\texttt{P} - \texttt{D}$ fra le spiagge successive a u_h che rispettano la disuguaglianza voluta. Supponiamo inoltre che $u_j \leq v \leq u_{j+1}$ per qualche $h \leq j < k$. Osserviamo allora che

$$\text{calura}(u_1, \ldots, u_h, v, u_{j+1}, \ldots, u_k) \leq \text{calura}(u_1, \ldots, u_k) \ ,$$

per cui anche $u_1, \ldots, u_h, v, u_{j+1}, \ldots, u_k$ sarebbe una scelta di spiagge ottima.

Questo vale perché gli unici termini che compaiono nel massimo che definisce $\text{calura}(u_1, \ldots, u_h, v, u_{j+1}, \ldots, u_k)$ e non in quello che definisce $\text{calura}(u_1, \ldots, u_k)$ sono $\text{dist}(u_h, v)$ e $\text{dist}(v, u_{j+1})$. D'altra parte $\text{dist}(u_h, v) \leq \text{dist}(u_h, u_{h+1})$ e $\text{dist}(v, u_{j+1}) \leq \text{dist}(u_h, u_{h+1})$, quindi i due termini extra di $\text{calura}(u_1, \ldots, u_h, v, u_{j+1}, \ldots, u_k)$ sono maggiorati da termini in $\text{calura}(u_1, \ldots, u_k)$, il che conclude la dimostrazione. □

In particolare data una generica spiaggia u, chiamiamo `next[`u`]` l'indice della spiaggia che minimizza `P[`v`]` + `D[`v`]` fra quelle per cui:

- $u < v$;
- `P[`v`]` − `D[`v`]` < `P[`u`]` − `D[`u`]`.

Allora questi due lemmi ci dicono come Mojito deve scegliere la prossima spiaggia in cui fermarsi se ora si trova nella spiaggia di indice u e sta percorrendo una soluzione ottima: in particolare esiste una soluzione ottima per cui Mojito si ferma nella spiaggia `next[`u`]` come prossima spiaggia.

Per trovare una scelta ottima di spiagge ci basta quindi partire dalla spiaggia fittizia u_1 all'inizio del lungomare e spostarsi alla spiaggia $u_2 =$ `next[`u_1`]`, poi alla spiaggia $u_3 =$ `next[`u_2`]` e così via fino ad arrivare alla fine del lungomare.

Quindi l'unico problema che rimane sta nel trovare *velocemente* la spiaggia `next`. In particolare troveremo `next[`u`]` per ogni spiaggia u, comprese quelle fittizie iniziali e finali, a partire dall'ultima a scendere fino alla prima utilizzando uno *stack*.

Costruiamo quindi uno stack `stack_next` che arrivati alla spiaggia u contiene degli indici $u_1, \ldots, u_k$ che rispettano le seguenti proprietà:

- $u < u_1 < \ldots < u_k$;
- u_1 minimizza `P` + `D` fra le spiagge successive a u;
- $u_{i+1} =$ `next[`u_i`]`, per ogni $i = 1, \ldots, k-1$.

Notiamo innanzitutto che non può mai accadere che `P[`u`]` − `D[`u`]` ≤ `P[`u_1`]` − `D[`u_1`]` e `P[`u`]` + `D[`u`]` ≥ `P[`u_1`]` + `D[`u_1`]`, poiché sommando le due disuguaglianze si otterrebbe `D[`u`]` ≥ `D[`u_1`]`, che contraddirebbe il fatto che $u < u_1$.

Di conseguenza, per quanto detto precedentemente se `P[`u`]` − `D[`u`]` ≤ `P[`u_1`]` − `D[`u_1`]`, allora sicuramente u_1 non potrà essere `next[`u`]`, ma non potrà essere nemmeno il `next` di nessun'altra spiaggia perché u sarà sicuramente migliore visto che `P[`u`]` + `D[`u`]` < `P[`u_1`]` + `D[`u_1`]`. In questo caso togliamo dunque u_1 da `stack_next` e ripetiamo la ricerca di `next` con il nuovo stack.

Se invece `P[`u`]` − `D[`u`]` > `P[`u_1`]` − `D[`u_1`]`, allora per le considerazioni precedenti $u_1 =$ `next[`u`]`. A questo punto aggiungiamo u a `stack_next` solo se `P[`u`]` + `D[`u`]` < `P[`u_1`]` + `D[`u_1`]`, altrimenti u_1 sarebbe un candidato migliore di u e potremmo quindi scordarci quest'ultima spiaggia.

Poiché ogni spiaggia entra ed esce al più una volta da `stack_next` e estrazione ed inserimento sono fatti solo dalla testa dello stack con un controllo in $O(1)$, l'algoritmo ha complessità $O(N)$.

Esempio di codice C++11

```cpp
#include <stack>
#include <cstdlib>

long long percorri(int N, long long L, long long D[], long long P[]) {

    // stack_next è uno stack che contiene le coppie (P+D, P-D)
    std::stack < std::pair <long long, long long> > stack_next;

    // Coppia corrispondente alla spiaggia fittizia a fine lungomare (D = L, P = 0)
    stack_next.push(std::make_pair(L, -L));

    for (int u = N-1; u >= 0; u--) {
        while (true) {
            if (P[u] - D[u] <= stack_next.top().second)
                stack_next.pop();
            else {
                if (P[u] + D[u] < stack_next.top().first)
                    stack_next.push(std::make_pair(P[u] + D[u], P[u] - D[u]));
                break;
            }
        }
    }

    // Modifico lo stack in modo da partire dall'inizio del lungomare (D = 0, P = 0)
    while (stack_next.top().second >= 0) stack_next.pop();
    stack_next.push(std::make_pair(0,0));

    long long res = 0;
    std::pair <long long, long long> prec = stack_next.top();
    std::pair <long long, long long> at;
    stack_next.pop();

    while (!stack_next.empty()) {
        at = stack_next.top();
        stack_next.pop();
        // at.first + prec.second è la distanza fra lido attuale e quello precedente:
        // (P[u_{i-1}]-D[u_{i-1}])+(P[u_i]+D[u_i])
        res = std::max(res, at.first + prec.second);
        prec = at;
    }

    return res;
}
```