

Pelstra di algoritmi

Marini Mattia

20 ottobre 2025

Palestra di algoritmi is licensed under [CC BY 4.0](#) .

© 2023 [Mattia Marini](#)

Indice

1	Introduzione	2
1.1	Basi cpp	2
1.1.1	Input metodo 1 (consigliato)	2
1.1.2	Input metodo 2	2
1.1.3	Ultra fast io	3
1.2	Complessità	4
1.2.1	Esempio 1	4
1.2.2	Esempio 2	5
1.3	Struttura problemi	6
2	Programmazione dinamica	7
2.1	Donimo	7
2.2	Hateville	7
2.3	Zaino	8
2.4	Zaino umbound	8
2.5	LCS	8
2.6	Occorrenza k approssimata	9
2.7	Prodotto di catena di matrici	10
2.8	Intervalli pesati	12
3	Problemi sito oii consigliati	12
4	Soluzioni problemi sito OII	13
4.1	Figonacci (figonacci)	13
4.2	Discesa massima (discesa)	13
4.3	Police 3 (police3)	14
4.4	Piano degli studi (pianostudi)	14
4.5	Police 4 (police4)	15
4.6	Spiedini di frutta (spiedini)	15
4.7	K-step ancestor (treeancestor)	16
4.8	Taglialegna (taglialegna)	16

1 Introduzione

Qui di seguito sono raccolte nozioni di base per affrontare ogni problema relativo alle *OII*

1.1 Basi cpp

In ogni problema è necessario effettuare input/output su file¹. Ci sono diversi modi per eseguire ciò.

1.1.1 Input metodo 1 (consigliato)

Vedi file `input1.cpp`

L'idea è di creare un oggetto `ifstream` e `ofstream` che poi potremmo utilizzare in maniera totalmente analoga a, rispettivamente, `cin` e `cout`

```
std::ifstream in("input.txt");
in >> a >> b;

std::ofstream out("output.txt");
out << a << b
```

Esiste un trucco per velocizzare notevolmente la velocità di input/output utilizzando questo metodo. In particolare, è sufficiente appendere le seguenti righe prima di scrivere o leggere su files:

```
ios_base::sync_with_stdio(false);
cin.tie(NULL);
```

Tuttavia se il problema sfora i limiti di tempo, con ogni probabilità è la soluzione a non essere corretta, non le operazioni di input/output. Queste righe possono essere utili per scalare la classifica sui siti di allenamento, non per altro

1.1.2 Input metodo 2

Vedi file `input2.cpp`

Questo metodo è più "vecchio" e meno consigliato. L'idea è di utilizzare le funzioni `freopen` per reindirizzare lo standard input/output su file:

```
FILE *in = fopen("input.txt", "r");
fscanf(in, "%d %d", &a, &b);

FILE *out = fopen("output.txt", "w");
fprintf(out, "%d %d\n", a, b);
```

dove le funzioni `fprintf` e `fscanf` prendono come argomenti:

- Il puntatore ad un file `FILE *`
- Una stringa `format`, contenente una serie di specificatori, preceduti da `"%"`

¹In realtà a volte è sufficiente implementare il body di una funzione oppure la parte relativa all'output viene fornita

- d: decimal, numero intero
 - f: float
 - s: stringa c-style, in particolare `char *`
- Una serie variabili che corrispondono a quanto indicato in `format`. Nel caso di `scanf` è richiesto l'indirizzo di memoria di queset

1.1.3 Ultra fast io

Ci sono infine alcuni metodi per velocizzare l'input al massimo, utili per spremere la performance al massimo, per arrivare nei primi in classifica. In particolare, questi metodi si basano sull'uso delle funzioni `getchat_unlocked()` e `putchar_unlocked()`

```
inline static int scanInt(FILE *file = stdin) {
    int n = 0;
    int neg = 1;
    char c = getc_unlocked(file);
    if (c == '-')
        neg = -1;
    while (c < '0' || c > '9') {
        c = getc_unlocked(file);
        if (c == '-')
            neg = -1;
    }
    while (c >= '0' && c <= '9') {
        n = (n << 3) + (n << 1) + c - '0';
        c = getc_unlocked(file);
    }
    return n * neg;
}

inline static void writeInt(int v, FILE *file = stdout) {
    static char buf[14];
    int p = 0;
    if (v == 0) {
        putc_unlocked('0', file);
        return;
    }
    if (v < 0) {
        putc_unlocked('-', file);
        v = -v;
    }
    while (v) {
        buf[p++] = v % 10;
        v /= 10;
    }
    while (p--) {
        putc_unlocked(buf[p] + '0', file);
    }
}
```

```

inline static int getString(char *buf, FILE *file = stdin) {
    std::string s;
    int c = getc_unlocked(file);

    // Skip leading whitespace
    while (c != EOF && (c == ' ' || c == '\n' || c == '\t' || c == '\r'))
        c = getc_unlocked(file);

    // Read until next whitespace or EOF
    int index = 0;
    while (c != EOF && c != ' ' && c != '\n' && c != '\t' && c != '\r') {
        buf[index++] = static_cast<char>(c);
        c = getc_unlocked(file);
    }

    return index;
}

inline static void putString(const std::string &s, FILE *file = stdout) {
    for (size_t i = 0; i < s.size(); i++)
        putc_unlocked(s[i], file);
}

```

Nota che le funzioni `putc_unlocked` e `getc_unlocked` sono disponibili solo in sistemi operativi unix (MacOs e Linux). Si possono usare in tranquillità dato che i server che testano il nostro codice sono tutti linux, ma il codice potrebbe non compilare in locale

1.2 Complessità

Il punto focale delle olimpiadi di informatica è non solo quello di scrivere algoritmi funzionanti, bensì efficienti. Per questa ragione è importante fornire critesi secondo i quali valutare la velocità d'esecuzione degli algoritmi

La logica di base sta nel relazionare il *numero di iterazioni* che un algoritmo deve eseguire alla *dimensione dell'input*.

1.2.1 Esempio 1

Supponiamo di avere un algoritmo per trovare il massimo in un vettore di n elementi. L'algoritmo fa quanto segue:

- Inizializza una variabile **max** al primo elemento del vettore
- Per ogni elemento del vettore controlla se è maggiore di **max**. In caso affermativo aggiorna **max** all'elemento corrente
- Ritorna **max**

Algoritmo: *Massimo vettore*

```
int max(int v[]):  
    max ← v[0];  
    for i = 0 to v.size - 1 do  
        if v[i] > max then  
            max ← v[i];  
    return max;
```

In questo caso notiamo come siano necessarie n iterazioni perchè l'algoritmo termini (dove n è la dimensione del vettore v). Abbiamo quindi rapportato la dimensione dell'input alla complessità temporale dell'algoritmo

In questo caso, si dice che la complessità dell'algoritmo è $\Theta(n)$

1.2.2 Esempio 2

Supponiamo di avere un algoritmo che debba eseguire una moltiplicazione applicando la proprietà distributiva:

$$(a + b + c) \cdot (d + e + f)$$

secondo la proprietà distributiva questo diventa:

$$\underbrace{(ad + ae + af)}_A + \underbrace{(bd + be + bf)}_B + \underbrace{(cd + ce + cf)}_C$$

ritornare un vettore che contenga i coefficienti (A, B, C)

Algoritmo: *Moltiplicazione distributiva*

```
int mul(int v1[], int v2[]):  
    int rv = int[0...v1.size];  
    for i = 0 to v1.size - 1 do  
        rv[i] = 0;  
        for j = 0 to v2.size - 1 do  
            rv+ = v1[i] · v2[j];  
    return rv;
```

Siccome per ogni elemento di v_1 devo scorrere interamente v_2 , dovrò ripetere $v_2 * v_1$ volte il body del ciclo.

In questo caso, se i due vettori hanno dimensione n , si dice che la complessità dell'algoritmo è $\Theta(n^2)$

1.2.2 Notazione Ω , Θ , O

In generale, per valutare la complessità di un algoritmo siamo interessati a più scenari:

- Nel peggiore dei casi, l'algoritmo che complessità ha? \rightarrow notazione O

- Nel migliore dei casi, l'algoritmo che complessità ha? \rightarrow notazione Ω
- Nel "caso medio", l'algoritmo che complessità ha? \rightarrow notazione Θ

Nota bene: nella maggior parte dei casi siamo interessati alla complessità nel caso pessimo O in quanto non possiamo escludere che questo si presenti nel dataset.

Per capire meglio la differenza fra caso ottimo e caso pessimo prendiamo in analisi l'algoritmo di *insertion sort*:

Algoritmo: *Insertion Sort*

```
int insertionSort(int v[]):
    for i = 1 to v.size - 1 do
        int key = v[i];
        int j = i - 1;
        while j ≥ 0 and v[j] > key do
            v[j + 1] = v[j];
            j = j - 1;
        v[j + 1] = key;
    return v;
```

In questo caso, dato un vettore lungo n , abbiamo due casi estremi:

- Il vettore è ordinato in modo crescente
- Il vettore è ordinato in modo decrescente

Nel primo caso l'algoritmo non entrerà mai nel ciclo while e dunque scorrerà il vettore una singola volta, originando una complessità di $\Omega(n)$.

Nel secondo caso l'algoritmo dovrà per ogni elemento del vettore scorrere (quasi) tutto il vettore stesso, originando una complessità di $O(n^2)$

1.3 Struttura problemi

Ogni problema delle OII e delle OIS ha una struttura simile e si compone come segue:

- Descrizione problema
- Descrizione dati di input
- Descrizione formato output
- Esempi
- Testcase

In particolare, il punteggio viene assegnato in base ai testcase che il nostro codice passa. Dobbiamo quindi scrivere un codice che risolva un dato problema stampando in output la soluzione. La correzione funziona come segue:

- I testcase sono raggruppati in un dato numero di *gruppi*
- Ad ogni gruppo di *testcase* è assegnato un punteggio e delle assunzioni. Ad esempio, ci può essere detto che i dati in input, in un dato gruppo non superano una certa dimensione o sono strutturati in un modo particolare
- Se all'interno di un gruppo i testcase sono tutti passati (output corretto), allora vengono assegnati i punti, altrimenti no

Si noti che per passare un testcase non è sufficiente che l'output sia corretto, ma il tempo di esecuzione e la memoria utilizzata devono essere entro i limiti previsti, specificati nel testo del problema

2 Programmazione dinamica

2.1 Donimo

Quanti modi ho di disporre tasselle di domino in una scacchiera $2 \times n$?

Soluzione

- Salvo in $dp[i]$ il numero di combinazioni che ci sono per un rettangolo $2 \times i$
- Ho due opzioni:
 - Metto 2 tessere in orizzontale, allora $dp[i] = dp[i - 2]$
 - Metto 1 tessera in verticale, allora $dp[i] = dp[i - 1]$
- Quindi $dp[i] = dp[i - 1] + dp[i - 2]$
- La soluzione è $Fib(n)$

2.2 Hateville

Ho un vettore di prezzi. Se prendo un prezzo $v[i]$ non posso prendere $v[i - 1]$ e $v[i + 1]$. Trova prezzo massimo

Soluzione

- Salvo in $dp[i]$ il prezzo massimo che posso ottenere con i vicini $\leq i$
- Ho due opzioni:
 - Non prendo $v[i]$, allora il prezzo è $dp[i - 1]$
 - Prendo $v[i]$, allora il prezzo è $dp[i - 1] + v[i]$

2.3 Zaino

Zaino ha capacità C , ho n pezzi di peso $w[i]$ e profitto $p[i]$. Trova profitto massimo

Soluzione

- Crea matrice $n \times C$ in cui si salva $dp[i][j]$ il profitto massimo che si può ottenere con i pezzi $\leq i$ e capacità $\leq j$
- Ho due opzioni:
 - Prendo pezzo (i, j) , allora il prezzo migliore è $dp[i-1][j-w[i]] + p[i]$
 - Non lo prendo, allora il prezzo è $dp[i-1][j]$
- Posso ottimizzare lo spazio tenendo salvato solo due righe della matrice, la i e la $i-1$

2.4 Zaino unbound

Vedi **zaino**, solo che non c'è limite al numero di oggetti che uno può prendere

Soluzione

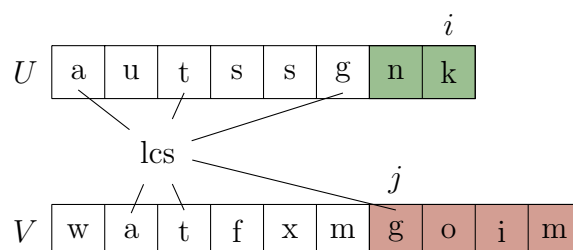
- Vettore dp in cui salvo in i il profitto massimo per uno zaino grande i
- Per ogni peso item x , il profitto massimo è $p[x] + dp[i-w[x]]$
- $dp[i]$ è il massimo fra tutti i valori trovati al punto 2

2.5 LCS

Date due stringhe U e T , trova la sottosequenza massimale. Una sottosequenza è una stringa che si ottiene da un'altra selezionandone solo alcuni caratteri (non necessariamente contigui, ma mantenendone l'ordine).

Soluzione

- Tabella dp con U su un lato e T sull'altro. In $dp[i][j]$ salvo la lunghezza della *LCS* fra la sottostringa $U[0, i]$ e $T[0, j]$
- Ho due opzioni:
 - $U[i] = T[j]$, allora $dp[i][j] = dp[i-1][j-1] + 1$ (aggiungo un carattere alla LCS più corta di 1)
 - $U[i] \neq T[j]$ allora $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$. Vedi immagine



Per migliorare la soluzione, se i caratteri sono diversi, devo aggiungere un carattere che sia nell'insieme dei caratteri dopo l'ultimo carattere comune. Quindi ho che

- A T , devo aggiungere un carattere che appartiene all'insieme rosso
- A U , devo aggiungere un carattere che appartiene all'insieme verde

Chiaramente la cosa è asimmetrica, per questo devo controllare $dp[i-1][j]$ e $dp[i][j-1]$

Dimostrazione formale : dobbiamo dimostrare che date due parole $U(u_1, \dots, u_i)$ e $V(v_1, \dots, v_j)$ e $X(x_1, \dots, x_k)$ allora

- Se $u_i = v_j$ allora

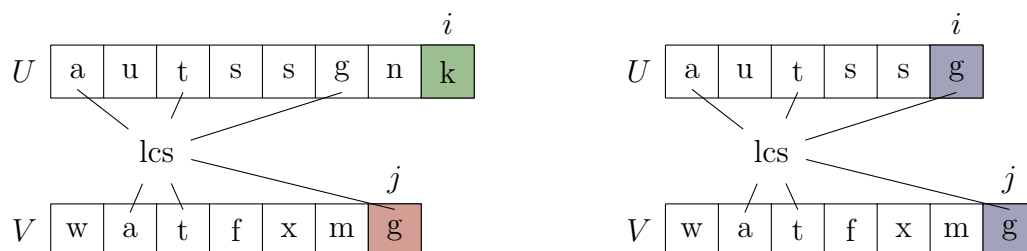
$$u_i = v_j = x_k \\ X(K-1) \in \mathcal{LCS}(U(i-1), V(j-1))$$

- Se $u_i \neq v_j$ e $x_k \neq u_i$ allora

$$X \in \mathcal{LCS}(U(i-1), V)$$

- Se $u_i \neq v_j$ e $x_k \neq v_j$ allora

$$X \in \mathcal{LCS}(U, V(j-1))$$



2.6 Occorrenza k approssimata

Data una stringa t e una p , diciamo che la distanza k di p da t è il numero minimo di *inserimenti*, *eliminazioni* e *scambi* che dobbiamo fare in t per far sì che $t == p$.

$$t = \text{"scempio"} , \quad p = \text{"esempio"} \rightarrow k = 2$$

ad esempio, scambiando la "s" e "c" di *scempio* in "e" ed "s" rispettivamente

Il problema sta nel trovare in un testo t , la distanza minima di un pattern p da una sua qualsiasi sottostringa.

Ciò equivale a trovare quanti inserimenti, rimozioni e scambi devo fare nel testo per far sì che il pattern diventi una sua sottostringa

Soluzione

- Inizializza matrice che ha p in verticale e t in orizzontale

- In $dp[i][j]$ si salva il minor valore di k per far sì che $p[0, i]$ sia sottostringa di $t[0, j]$ che finisca in j
- Se $p[i] == t[j]$ allora non serviranno altre mosse per riportare la soluzione di $dp[i-1][j-1]$ alla soluzione corrente
- Se $p[i] \neq t[j]$ allora posso fare 3 cose:

	a	b	a	b	a	g
b						
a						
b		→	?			

+1
Fai coincidere bab con ab
e poi elimina a

	a	b	a	b	a	g
b						
a			↓			
b			?			

+1
Fai coincidere ba con aba
e poi aggiungi b

	a	b	a	b	a	g
b						
a		↘				
b			?			

+1
Fai coincidere ba con ab
e poi cambia a in b

La soluzione migliore è data dal minimo valore nell'ultima riga della tabella

Nota che la prima riga e la prima colonna vanno riempite rispettivamente con $[0, \dots, 0]$ e $[1, 2, \dots, n-1, n]$. Questo ha senso in quanto:

- Per far sì che il pattern vuoto sia sottostringa di t non serve alcuna mossa ($[0, \dots, 0]$)
- Per far sì che un pattern di lunghezza k sia sottostringa del testo vuoto è necessario aggiungere i k caratteri del pattern ($[1, 2, \dots, n-1, n]$)

2.7 Prodotto di catena di matrici

Si vuole fare il prodotto matriciale tra $[A_1, A_2, \dots, A_{n-1}, A_n]$. Il prodotto matriciale gode di proprietà associativa. Si trovi la parentizzazione che riduce al minimo il numero di moltiplicazioni scalari totali da compiere

Ad esempio, avendo $[A, B, C, D]$, posso parentizzare come segue:

$$[(A \cdot B) \cdot (C \cdot D)], \quad [A \cdot (B \cdot C) \cdot D], \quad [A \cdot (B \cdot (C \cdot D))]$$

e così via. Questo funziona in quanto per moltiplicare delle matrici bisogna assicurarsi che queste siano compatibili. Il numero di colonne della prima deve essere uguale al numero di righe della seconda. Ad esempio, indicando con $[righe, colonne]$ una matrice, una serie che può essere moltiplicata è la seguente:

$$[4, 5] \cdot [5, 2] \cdot [2, 10] \cdot [10, 7] \rightarrow [4, 5, 2, 10, 7]$$

Nota che la dimensione di ogni matrice può essere salvata in un vettore c in cui c_i contiene il numero di colonne della matrice i , che corrisponde al numero di righe della matrice $i+1$. Quindi il numero di moltiplicazioni necessarie per eseguire $A_i \times A_j$ sarà:

$$c_i \cdot (c_{i-1} \cdot c_j)$$

- c_i : numero di moltiplicazioni per calcolare una cella

- $(\cdot c_{i-1} \cdot c_j)$: dimensione della matrice risultante

Soluzione

- Creo matrice **dp** come seguen:

	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

- In **dp[i][j]** salvo il minor numero di moltiplicazioni necessarie per moltiplicare le matrici fra **i** e **j**
- Costruisco matrice scorrento in diagonale a partire dalla diagonale più vicina alla diagonale principale. Il numero minore è dato dal numero minore date due parentizzazioni, ad esempio se ho

$$[A_3, A_4, A_5, A_6]$$

dovro tentare con

$$[(A_3) \cdot (A_4, A_5, A_6)], \quad [(A_3, A_4) \cdot (A_5, A_6)], \quad [(A_3, A_4, A_5) \cdot (A_6)]$$

- Il risultato finale si trova in **dp[1][n]**, dove **n** è il numero di matrici
- Per ricostruire la parentizzazione, posso salvarmi in una tabella **last[i][j]** l'indice a cui ho "spezzato la parentizzazione". Poi posso ricostruirla ricorsivamente come segue:

Algoritmo 1: *Find minimum parenthesization*

```

printPar(int last[], int i, int j):
    if i == j then
        print "A["; print i; print "];"
    else
        print "(";
        printPar(last, i, last[i][j]);
        print ".";
        printPar(last, last[i][j] + 1, j);
        print ");"
```

Algorithm 1: Print optimal parenthesization

2.8 Intervalli pesati

Vengono dati n intervalli aperti $[a_1, b_1[, [a_2, b_2[, \dots, [a_n, b_n[$. Ogni intervalli ha un valore w_i . Trovare il valore massimo che si può ottenere selezionando intervalli non sovrapposti.

Soluzione

- Ordina intervalli per tempo di fine
- Definisco la funzione `pred(i)`, che ritorna il *predecessore* di un intervallo, ossia il primo intervallo che ha tempo di fine minore del tempo di inizio di i
- Creo vettore `dp` che salva in i il valore massimo ottenibile con gli intervalli fino ad i compreso
- Itero su intervalli. Per ciascun intervallo i posso:
 - Selezionarlo: in questo il valore massimo ottenibile è dato da `dp[pred(i)] + wi` a
 - Non selezionarlo: in questo caso il valore massimo è uguale al precedente `dp[i-1]`

Complessità: $O(n \log n)$

3 Problemi sito oii consigliati

Di seguito una raccolta di problemi provenienti dal sito degli allenamenti della OII. Sono proposte delle soluzioni in sezione [sezione 4](#)

Problema	Tecniche	Difficoltà	Soluzione
Fibonacci (fibonacci)	dp	★ ☆ ☆ ☆ ☆	sezione 4.1
Discesa massima (discesa)	dp	★ ☆ ☆ ☆ ☆	sezione 4.2
Police 3 (police3)	dp	★ ☆ ☆ ☆ ☆	sezione 4.3
Piano degli studi (pianostudi)	dp, binary search	★ ★ ★ ☆ ☆	sezione 4.4
Spiedini di frutta (spiedini)	dp	★ ★ ☆ ☆ ☆	sezione 4.6
K-step ancestor (treeancestor)	dp, graph	★ ★ ☆ ☆ ☆	sezione 4.7
Taglialegna (taglialegna)	dp, amortized analysis	★ ★ ★ ★ ★	sezione 4.8

4

Soluzioni problemi sito OII

Di seguito una raccolta di soluzioni per i problemi proposti in [sezione 3](#). Per ogni problema è riportato il link alla pagina del problema e il link alla soluzione proposta in questa dispensa.

4.1 [Fibonacci \(fibonacci\)](#)

Problema sito OII

https://training.olinfo.it/task/ois_fibonacci

Soluzione proposta

<files/esercizi/fibonacci>

L'idea qui è di "srotolare" la sommatoria, rendendosi conto che

$$\begin{aligned}
 G_n &= \sum_{i=0}^{n-2} G_{n-1} - G_i \\
 &= (G_{n-1} - G_{n-2}) + (G_{n-1} - G_{n-3}) + \dots + (G_{n-1} - G_1) + (G_{n-1} - G_0) \\
 &= (n-1) G_{n-1} - \sum_{i=0}^{n-2} G_i \\
 &= (n-1) G_{n-1} + G_{n-1} - (n-2) G_{n-2} - G_{n-2} \\
 &= (n-1) G_{n-1} - (n-1) f(n-2) + G_{n-1} \\
 &= (n-1) (G_{n-1} - G_{n-2}) + G_{n-1}
 \end{aligned}$$

abbiamo così ottenuto a tutti gli effetti una formula che possiamo applicare direttamente in forma ricorsiva (con *memoization*) o iterativa:

$$dp[i] = (i-1)(dp[i-1]*dp[i-2]) + dp[i-1]$$

4.2 [Discesa massima \(discesa\)](#)

Problema sito OII

<https://training.olinfo.it/task/discesa>

Soluzione proposta

<files/esercizi/discesa>

L'idea in questo caso è di creare un albero di discesa in cui in ogni posizione $[i][j]$ salviamo *il maggior peso di una discesa che termina nella posizione $[i][j]$* .

Ad esempio, qui sotto è riportato a destra l'albero dp per l'albero in input di sinistra

$$\begin{array}{ccccc}
 & 1 & & & 1 \\
 & 2 & 9 & & 3 & 10 \\
 & 3 & 7 & 5 & \rightarrow & 6 & 17 & 15 \\
 8 & 4 & 11 & 6 & & 14 & 21 & 28 & 21
 \end{array}$$

rimane da capire come calcolare i valori nella file i esima basandosi sui precedenti. In particolare abbiamo 3 casi:

- *Primo elemento*: posso arrivare solo da destra, quindi:

$$dp[i][0] = dp[i-1][0] + w[i][0]$$

- *Ultimo elemento*: posso arrivare solo da sinistra, quindi

$$dp[i][size] = dp[i-1][size-1] + w[i][size]$$

- *Elemento in mezzo*: posso arrivare sia da sinistra che da destra, quindi dovrò prendere la strada che fra le due ha peso maggiore:

$$dp[i][j] = w[i][j] + \max(dp[i][j], dp[i][j+1])$$

Chiaramente il caso base è quello riguardante il primo elemento, per cui il percorso di peso massimo è dato dal peso dell'elemento stesso.

Occorre ragionare un filo meglio sugli indici ma la logica rimane inalterata. Nota bene: *è veramente necessario salvare l'intera matrice dp?*

4.3 Police 3 (police3)

Problema sito OII	https://training.olinfo.it/task/ois_police3
Soluzione proposta	files/esercizi/police3

L'idea qui è, per quanto banale possa sembrare, ad ogni semaforo abbiamo due opzioni: fermarsi o meno. Per questa ragione possiamo salvare informazioni in due vettori **dp**, per ciascuno dei due casi:

- **ts[i]**: salva in *i* il minor tempo per superare il semaforo *i* fermandosi a quest'ultimo
- **tr[i]**: salva in *i* il minor tempo per superare il semaforo *i*, scappando

possiamo costruire i valori dei due vettori basandoci sui precedenti secondo la seguente logica

- Se *mi fermo* al semaforo *i*-esimo, il tempo ideale sarà dato dall'attesa al semaforo + il tempo migliore ottenuto fermandosi o meno a quello precedente (posso scegliere di fermarmi oppure no, siccome non violo nessun vincolo in ogni caso)

$$ts[i] = T[i] + \min(tr[i-1], ts[i-1]);$$

- Se *non mi fermo* al semaforo *i*-esimo, il tempo ideale sarà dato dal tempo ideale per arrivare al semaforo *i-1*, fermandocisi (in questo caso devo assicurarmi di fermarmi al semaforo precedente per non violare alcun vincolo)

$$tr[i] = ts[i-1];$$

Anche in questo caso, è davvero necessario salvare l'interi vettori in memoria?

4.4 Piano degli studi (pianostudi)

Problema sito OII	https://training.olinfo.it/task/ois_pianostudi
Soluzione proposta	files/esercizi/pianostudi

4.5 Police 4 (police4)

Problema sito OII

https://training.olinfo.it/task/ois_police4

Soluzione proposta

<files/esercizi/police4>

L'idea in questo caso è di salvare di volta il tempo minimo per arrivare al semaforo i avendone saltati al più j . Questa informazione può essere mantenuta in una matrice $n \times r$, dove n è il numero di *semafori* e r il numero massimo di semafori che *possono essere saltati*.

La i -esima colonna/riga della matrice può essere calcolata usando la precedente. In particolare, supponiamo di voler calcolare il tempo minore per arrivare al semaforo i saltandone al più j . Allora ho due opzioni:

- Arrivo al semaforo $i - 1$ saltandone al massimo $j - 1$. In questo caso il tempo ideale è quello ottenuto saltando il semaforo $i - 1$
- Arrivo al semaforo $i - 1$ saltandone al più j . In questo caso devo controllare se devo aspettare o meno. In caso affermativo devo aggiungere il tempo di attesa alla soluzione

La soluzione ottimale è data dalla migliore delle due. Formalmente dati T e X , rispettivamente l'intervallo dei semafori e il vettore delle posizioni, posso calcolare dp come segue:

```
t1 = dp[i-1][j-1];
t2 = dp[i-1][j] + (can_pass ? 0 : T-dp[i-1][j] % T);
dp[i][j] = min(t1, t2) + X[i] - X[i-1];
```

4.6 Spiedini di frutta (spiedini)

Problema sito OII

https://training.olinfo.it/task/oii_spiedini

Soluzione proposta

<files/esercizi/spiedini>

L'idea è che una volta che uno spiedino è stato mangiato fino ad un certo punto da una parte, allora la soluzione ideale è prefissata e consiste nel mangiarlo dalla parte opposta finché è possibile. La chiave sta tuttavia nel comprendere che non è necessario calcolare tutta la somma da capo ad ogni iterazione, ma possiamo seguire questa logica:

- Mangio lo spiedino *da sinistra* fino all'ultima fragola che permette di *non* superare la soglia. Tengo sempre traccia della soglia corrente
- Partendo *da destra* mangio lo spiedino fino a quando la soglia lo permette
- Tengo traccia del numero di fragole mangiate i

Dopodichè, ripeto questi step fino a quanto non raggiungo l'estremità sinistra dello spiedino:

- Da sinistra, scorro fino alla fragola precedente, abbassando la soglia
- Da destra scorro fino all'ultima fragola che riesco ad includere, aggiornando soglia e quantità di fragole mangiate

La soluzione migliore ottenuta in ogni step dell'iterazione precedente è la soluzione ottimale al problema

4.7 K-step ancestor ([treeancestor](#))

Problema sito OII	https://training.olinfo.it/task/ois_treeancestor
Soluzione proposta	files/esercizi/treeancestor

La soluzione sta nell'esplorare l'albero tenendo traccia della profondità corrente e del tragitto che ha portato fino al nodo che stiamo attualmente visitato. Possiamo fare ciò tramite 2 variabili `depth` e `path[n]`

Dato che il grado in questo caso è semplicemente un albero, possiamo sfruttare anche una semplice DFS, dato che possiamo essere sicuri che il path che ci porta a ciascun nodo è unico.

Per ciascun nodo, controlliamo se la `depth` è maggiore o minore di K . Nel primo caso sfruttiamo l'array contenente il path per assegnare il valore del k -ancestor, nel secondo semplicemente assegniamo -1. Formalmente:

```
if (depth >= K) {
    ancestor[node] = path[depth - K];
} else {
    ancestor[node] = -1;
}
```

nella DFS chiamiamo ricorsivamente la funzione con un valore di `depth` incrementato di 1 ogni volta

4.8 Taglialegna ([taglialegna](#))

Problema sito OII	https://training.olinfo.it/task/oii_taglialegna
Soluzione proposta	files/esercizi/taglialegna

Il problema è piuttosto avanzato, per questo è necessario fare delle osservazioni preliminari che torneranno utili dopo.

1. In una soluzione ottima ho 3 opzioni:
 - (a) L'ultimo albero a destra viene tagliato e fatto cadere a *sinistra*
 - (b) L'ultimo albero a destra viene tagliato e fatto cadere a *destra*
 - (c) L'ultimo albero a destra *non* viene tagliato ma fatto cadere a *destra* per "effetto domino"

L'idea importante è che in una soluzione ottima l'ultimo albero a destra, se cade a sinistra è perchè è stato tagliato e fatto cadere a sinistra. Per quanto sia ovvio è molto importante per capire gli step successivi

2. Gli unici punti in cui si può interrompere l'effetto domino sono i punti in cui vi sono alberi di altezza 1
3. Per quanto detto al punto precedente, possiamo vedere il problema come una serie di intervalli che errano fatti cadere in una delle due direzioni. Agli estremi di questi intervalli ci sono alberi di altezza 1
4. Quando avviene un effetto domino, non tutti gli alberi partecipano nella catena della caduta. In particolare un albero può essere abbattuto da un albero che non è quello precedente nella catena. Chiameremo questo albero *ff*(first-falling) di *i*. Possiamo quindi considerare l'effetto domino solo per gli alberi abbattitori, nel modo seguente:

$$i \rightarrow \text{ff}[i] \rightarrow \text{ff}[\text{ff}[i]] \rightarrow \dots$$

Detto questo, individuiamo il sotto-problema di programmazione dinamica e il caso base. In particolare sappiamo che:

- Il sotto-problema consiste nel calcolare il *il numero minore di tagli* per abbattere gli alberi da 0 a *i*
- Il caso base è quanto $n = 1$, ossia quando ho un solo albero. In questo caso la soluzione è fare un unico taglio

Rimane ora capire come utilizzare le soluzioni ottime da 0 a $i - 1$ per calcolare la *i*-esima soluzione ideale. Possiamo ragionare così per calcolare la soluzione ottimale *i*-esima:

- Se l'albero viene *tagliato a sinistra*, allora simulo l'effetto domino e verifico dove questo si interrompe. Supponendo che la catena abbia fatto cadere *k* alberi, allora

$$\text{solution}[i] = 1 + \text{solution}[i-k-1]$$

- Se l'albero viene *tagliato a destra*, allora non rimane che pulire in maniera ottimale gli alberi da 0 a $i - 1$, quindi:

$$\text{solution}[i] = 1 + \text{solution}[i-1]$$

- Se l'albero viene *fatto cadere a destra*, allora all'interno della *catena dei first-fall* devo capire quale albero è più conveniente tagliare per innescare l'effetto domino. Chiamando questo albero *opt*, allora la soluzione ottimale è data da

$$\text{solution}[i] = 1 + \text{solution}[\text{opt}-1]$$

Andiamo ora ad elencare le strutture dati che ci aiuteranno nel calcolo progressivo del vettore **solution**

- **lb[i]**: *left-bound*, contiene nella posizione *i* l'indice fino al quali gli alberi cadono se l'albero *i* viene tagliato a *sinistra*
- **rb[i]**: *right-bound*, contiene nella posizione *i* l'indice fino al quali gli alberi cadono se l'albero *i* viene tagliato a *destra*

- `ff[i]`: *first-fall*, come descritto precedentemente, il primo albero alla sinistra di i che in un effetto a catena colpisce i

Come vedremo, possiamo calcolare tutti questi vettori in tempo $O(n)$

- `solution[i]`: il vettore *dp* principale. Salva la soluzione ottimale per il range $[0, i]$
- `dir[i]`: il vettore che contiene la direzione in cui va tagliato il primo albero per ottenere la soluzione ottimale
- `opt_split[i]`: *optimal-split*, salva la posizione ottimale per dare inizio all'effetto domino nella soluzione migliore in cui l'ultimo albero cade a destra
- `first_cut[i]`: salva l'indice del primo albero da tagliare nella i -esima sotto soluzione ottimale

Parte 1: calcolo `lb`, `rb`

Partiamo dal calcolo dei vettori `lb` e `rb`. L'idea è proprio quella di simulare a tutti gli effetti l'effetto domino. In particolare, pensiamo di avere una serie di intervalli che cadono per effetto domino e un albero che viene fatto cadere a sinistra:

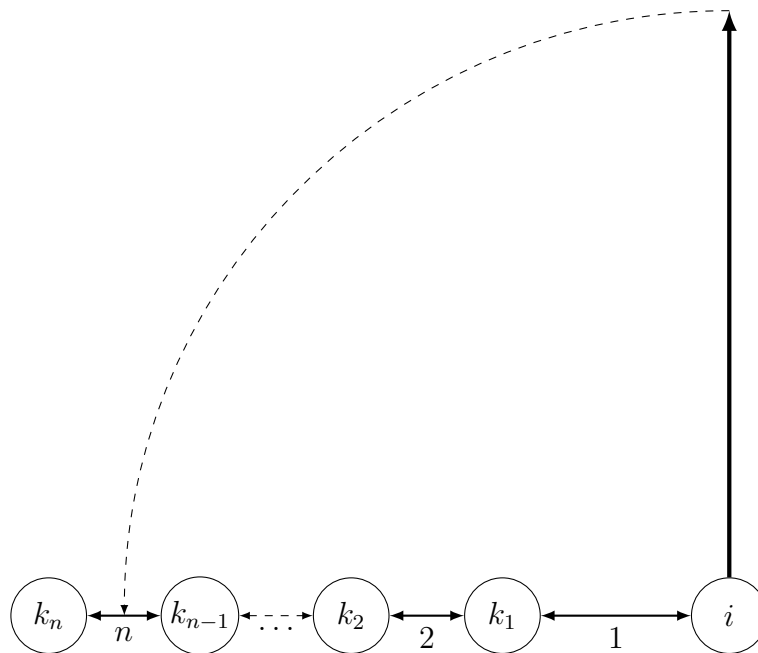


Figura 1: Calcolo left-bound

In questa figura, sappiamo già che gli intervalli da 1 a n cadono a sinistra per effetto domino, interrompendosi nei punti k_2, \dots, k_n . `lb[i]` sarà quindi dato da valore più basso fra i `lb` degli intervalli che l'albero abbatte sicuramente, ossia tutti gli intervalli il cui albero più a destra viene abbattuto dall'albero i . In figura, questi sono precisamente gli intervalli $1, \dots, n$. Più formalmente

Algoritmo: *Calcolo left-bound*

```
int [] compute_left_bound(n, h):  
  int lb ← new int [0...n-1];  
  for i ← 0 to n-1 do  
    lb[i] ← i;  
    while lb[i] > 0 ∧ lb[i] > i - h[i] + 1 do  
      lb[i] ← lb[lb[i] - 1];  
  return lb;
```

Il calcolo del vettore **rb** è speculare, ma segue la medesima logica.

Ora è fondamentale discurre la complessità di questa funzione. Potremmo in un primo momento pensare *erroneamente* che la funzione abbia complessità di $O(n^2)$, dato che il ciclo esterno scorre l'intero vettore e, nel caso peggiore, anche quello interno scorre tutti gli elementi da i a 0. Tuttavia, il ragionamento è un filo più sottile.

Diciamo che un albero viene *saltato da i* se all'interno del ciclo while che calcola **lb[i]** viene utilizzato per aggiornare il valore di *lower-bound*

$$lb[i] \leftarrow lb[\underbrace{lb[i] - 1}_{\text{saltato}}]$$

Aiutandoci con la figura 1, possiamo intuire che gli alberi saltati, nel calcolo di **lb[i]**, sono dati da k_1, k_2, \dots, k_{n-1} . La chiave sta nel capire che ogni albero può essere saltato una volta sola. Questo avviene in quanto, supponendo che gli alberi k_1, \dots, k_{n-1} vengano saltati dall'albero i , allora nelle iterazioni successive questi non possono più essere saltati, in quanto al più verrà saltato prima l'albero i , il quale avrà un *left-bound* minore di k_{n-1} , quindi tutto il range in cui stanno gli alberi k_1, \dots, k_{n-1} non verrà considerato nella computazione.

Quindi, ancora una volta, siccome ogni albero viene saltato una e una sola volta, la riga $lb[i] \leftarrow lb[lb[i] - 1]$ del ciclo while viene eseguita al più n volte, portando ad una complessità totale di $O(n)$

Parte 2: calcolo **ff**

Il calcolo di **ff** può implementato in maniera piuttosto semplice con piccolissime modifiche alla funzione che calcola **rb**. Sempre riferendoci a [fig. 1](#), possiamo notare che il primo albero ad abbattere gli alberi k_1, \dots, k_{n-1} in una catena di abbattitori è i . Per questo motivo ogni volta che calcoliamo **rb[i]** possiamo anche aggiornare il valore per **ff** degli alberi k_1, \dots, k_{n-1} . Formalmente:

Algoritmo: *Calcolo right-bound e first-fall*

```
int // compute_rb_ff(n, h):
    int rb ← new int [0...n - 1];
    int ff ← new int [0...n - 1];
    for i ← n - 1 downto 0 do
        rb[i] ← i;
        ff[i] ← -1;
        while rb[i] < n - 1 and rb[i] < i + h[i] - 1 do
            ff[rb[i] + 1] ← i;
            rb[i] ← rb[rb[i] + 1];
    return rb, ff;
```

La complessità rimane sempre $O(n)$

Parte 3: calcolo solution

Innanzitutto, ci appoggeremo alle strutture dati **dp** descritte [qui](#). Per calcolare il vettore **solution**, gli step sono, dal punto di vista logico, quelli descritti [qui](#). Descriviamo più nel dettaglio le 2 casistiche:

- Taglio a sinistra:

```
solution[i] = 1 + (lb[i] > 0 ? solution[lb[i] - 1] : 0);
opt_cut[i] = i;
dir[i] = 0;
```

- Albero cade a destra:

- Taglio a destra:

```
solution[i] = 1 + (idx2 > 0 ? solution[i - 1] : 0);
```

- Albero abbattuto a destra per effetto domino:

```
split_index = opt_split[ff[i]]; // Optimal index to cut the tree
solution[i] = 1 + (idx1 > 0 ? solution[split_index - 1] : 0);
```

Ora ci rimane solo da aggiornare il vettore **opt_cut**, così da rendere l'intero algoritmo efficiente. Anche qui la programmazione dinamica viene in nostro soccorso. Più in particolare, so che supponendo che la catena di k first fall che abbattano i sia composta dagli alberi $1, 2, \dots, k - 1, i$. Allora so già che fra i primi $k - 1$ il migliore è dato da **opt_split**[$k - 1$], non rimane che testare l'ultimo, ossia il caso in cui i è abbattuto a destra. Nel caso la soluzione migliore si ottenesse con la prima opzione, allora **opt_split**[i]=**opt_split**[**ff**[i]], altrimenti **opt_split**[i] = i

Chiaramente, scorrendo ogni elemento del vettore una sola volta, la funzione ha complessità $O(n)$

Algoritmo: *Calcolo soluzione*

```
compute_solution( $n, lb, rb, ff$ ):  
     $solution[0] \leftarrow 1$ ;  
     $dir[0] \leftarrow 1$ ;  
     $opt\_split[0] \leftarrow 0$ ;  
     $first\_cut[0] \leftarrow 0$ ;  
    for  $i \leftarrow 1$  to  $n$  do  
         $\triangleright$  Case 1: last tree is cut to the left  
         $sol_1 \leftarrow 1 + \text{if } lb[i] > 0 \text{ then } solution[lb[i] - 1] \text{ else } 0$ ;  
         $cut_1 \leftarrow i$ ;  
         $dir_1 \leftarrow \text{false}$ ;  
         $\triangleright$  Case 2: last tree falls to the right  
        if  $ff[i] \neq -1$  then  
             $\triangleright$  Case 2.1: tree falls by domino effect  
             $idx_1 \leftarrow opt\_split[ff[i]]$ ;  
             $sol_{21} \leftarrow 1 + \text{if } idx_1 > 0 \text{ then } solution[idx_1 - 1] \text{ else } 0$ ;  
             $\triangleright$  Case 2.2: tree is cut to the right  
             $idx_2 \leftarrow i$ ;  
             $sol_{22} \leftarrow 1 + \text{if } idx_2 > 0 \text{ then } solution[idx_2 - 1] \text{ else } 0$ ;  
             $opt\_split[i] \leftarrow \text{if } sol_{21} < sol_{22} \text{ then } idx_1 \text{ else } idx_2$ ;  
        else  
             $opt\_split[i] \leftarrow i$ ;  
         $sol_2 \leftarrow 1 + \text{if } opt\_split[i] > 0 \text{ then } solution[opt\_split[i] - 1] \text{ else } 0$ ;  
         $cut_2 \leftarrow opt\_split[i]$ ;  
         $dir_2 \leftarrow \text{true}$ ;  
        if  $sol_1 < sol_2$  then  
             $solution[i] \leftarrow sol_1$ ;  
             $first\_cut[i] \leftarrow cut_1$ ;  
             $dir[i] \leftarrow dir_1$ ;  
        else  
             $solution[i] \leftarrow sol_2$ ;  
             $first\_cut[i] \leftarrow cut_2$ ;  
             $dir[i] \leftarrow dir_2$ ;  
    return  $solution, first\_cut, dir$ ;
```

Parte 4: ricostruire la soluzione

Una volta calcolati i vettori **solution**, **first_cut** e **dir** è piuttosto immediato ricostruire la soluzione "risalendo" al contrario i tagli che sono stati fatti:

Algoritmo 2: Ricostruzione soluzione

```
reconstruct_solution( $n, first\_cut, dir$ ):  
   $pos \leftarrow n - 1$ ;  
  while  $pos \geq 0$  do  
    abbatti( $first\_cut[pos], dir[pos]$ );  
    if  $dir[pos] = \text{false}$  then  
       $pos \leftarrow lb[pos] - 1$ ;  
    else  
       $pos \leftarrow first\_cut[pos] - 1$ ;
```

Complessivamente, abbiamo che

Step	task	complessità
Step 1	calcolo <code>lb</code> , <code>rb</code>	$O(n)$
Step 2	calcolo <code>ff</code>	$O(n)$
Step 3	calcolo <code>solution</code> , <code>dir</code> , <code>best_cut</code>	$O(n)$
Step 4	ricostruisco soluzione	$O(n)$

Complessità totale: $O(n)$