Pelstra di algoritmi

Marini Mattia

20 ottobre 2025

Indice

1 Introduzione

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

1.1 Basi cpp

In ogni problema è necessario effettuare input/output su file[1]. Ci sono diversi modi per eseguire ciò.

1.1.1 Input metodo 1 (consigliato)

Vedi file input1.cpp

L'idea è di creare un oggeto `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::ifstream 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 "%"

[1] In 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 sprepere la perfomance 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 algorimo 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, dovro 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? → 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 maggio parte dei casi siamo interessati alla coplessità 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 compessità 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 $\text{Fib}(n)$

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 umbound

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

Soluzione

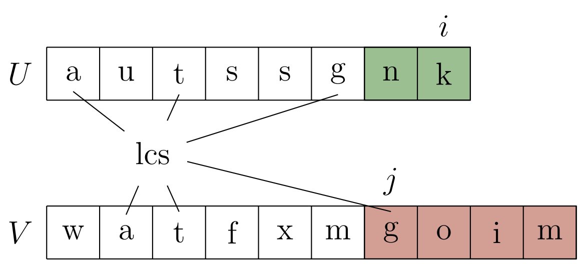
- Vettore dp in cui salvo in i il profitto massimo per uno zaino grande i
- Per ogni peso item x, il profitto massimo è $p[x] + dp[i - w[x]]$
- $dp[i]$ è il massimo fra tutti i valori trovati al punto 2

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 $dp\left[i-1\right]\left[j\right]$ e $dp\left[i\right]\left[j-1\right]$

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

- Se $u_i = v_j$ allora

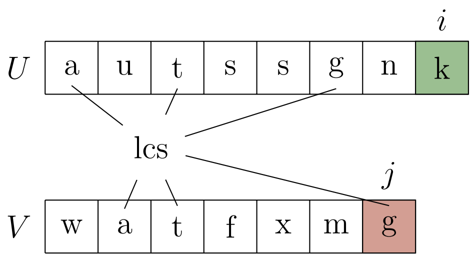
$$u_i = v_j = x_k$$
$$X\left(K-1\right) \in \mathcal{LCS}\left(U\left(i-1\right), V\left(j-1\right)\right)$$

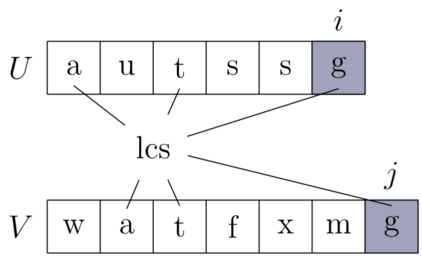
- Se $u_i \neq v_j$ e $x_k \neq u_i$ allora

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

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

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





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 si 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 problma 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 si 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 si 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 $[0,\ldots,0]$ e $[1,2,\ldots,n-1,n]$. Questo ha senso in quanto:

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

2.7 Prodotto di catena di matrici

Si vuole fare il prodotto matriciale tra $[A_1, A_2, \ldots, A_{n-1}, A_n]$. 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 cosi 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 $[\text{righe}, \text{colonne}]$ 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:

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

- 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 $[a_1, b_1[\,, [a_2, b_2[\,, \ldots [a_n, b_n[$. 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)] + w_i` a
  - Non selezionarlo: in questo caso il valore massimo è uguale al precedente `dp[i-1]`

Complessità: $O(n \log n)$

2.9 Problemi sito oii consigliati

- Figonacci (figonacci)
- Discesa massima (discesa)
- Police 3 (`police3`)
- Piano degli studi (pianostudi)
- Police 4 (`police4`)
- Spiedini di frutta (spiedini)