Pelstra di algoritmi

Marini Mattia

20 ottobre 2025

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 Problemi dp . . . 7
  - 2.1.1 Police 3 (`police3`) . . . 7

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? $\rightarrow$ notazione O

- Nel migliore dei casi, l'algoritmo che complessità ha?→ notazione Ω
- Nel "caso medio", l'algoritmo che complessità ha?→ 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 Problemi dp

2.1.1 Police 3 (`police3`)