Introduzione, Input/Output & Data Structures

Basato su Competitive Programmer's Handbook, Dispense del prof. Bugatti e Halim's Competitive Programming 3

Diteci qualcosa di voi

https://tinyurl.com/pda2425

Obiettivi

Gli obiettivi delle competizioni di programmazione sono

- Pensare a soluzioni efficienti per un problema
- Pensare velocemente a soluzioni
- Scrivere velocemente le soluzioni

È un gioco molto equo: più vi esercitate, migliori sarete in queste tre competenze. Trovate tanti esercizi a https://training.olinfo.it/

Perchè programmazione competitiva

Allenandosi in un ambiente competitivo, soprattutto a squadre, rende molto più stimolante imparare a pensare e scrivere bene e velocemente.

Le abilità che svilupperete qui vi consentiranno di saper scrivere programmi e software performanti, efficaci e testati, in un mondo di app poco ottimizzate e piene di bug.

Queste skill sono multidisciplinari: ad esempio, in biotecnologie o ingegneria è necessario processare grandi quantità di dati in poco tempo.

Prerequisiti

Il corso è tenuto in C++ e Python. Un linguaggio a basso livello vi permette di pensare meglio a strutture dati, gestione memoria ed efficienza di esecuzione. Tuttavia, usare un linguaggio di alto livello vi permette di pensare alla cosa più importante: l'algoritmo.
Conoscenze di base (che ripasseremo):

- tipi di dato base: int, char, array, boolean
- condizioni: if, else, switch
- cicli: for, while, do_while
- operazioni booleane: and, or, not, xor, nand

Utilizzo delle LLM, come ChatGPT

Le LLM moderne sono un ottimo strumento per professionisti per delegare mansioni noiose, ripetitive e ben conosciute.

Voi siete qui per imparare e sperimentare, non lavorare! Non fatevi rubare occasioni preziose per migliorare!

Tuttavia, le LLM sono anche un ottimo strumento per apprendere, se usate bene: soprattutto con quelle più moderne, scrivete subito che siete `ragazzi che stanno imparando a programmare`, e che vi serve `aiuto per superare un blocco`: questi tool sono in grado di aiutarvi a capire le cose molto più velocemente, se usate bene.

No ingegneria del software

Per motivi pratici e mancanza di tempo, userete tattiche sconsigliate se mai lavorerete a codice che necessita di essere mantenuto.

- variabili globali
- poche funzioni o modularità
- abbreviazioni di costrutti comuni
- scelta accurata delle librerie

Template base in C++

```cpp
#include <bits/stdc++.h> // tutte le librerie di C/C++, filtrate poi dal compilatore
using namespace std;

int N, K; // variabili input globali
int main() {
    ios::sync_with_stdio(0); // input/output più veloce
    cin.tie(0); // input/output più veloce

    freopen("input.txt", "r", stdin); // sovrascrivi il flusso di input da file
    freopen("output.txt", "w", stdout); // sovrascrivi il flusso di output su file

    cin >> N >> K; // salta spazi e "a capo" (newlines) fino a prossimo valore
    cout << N + K << "\n"; // "\n" è più veloce di endl (flush)
}
```

Template base in Python

```python
import sys

# Redirect input/output da/a file
sys.stdin = open('input.txt', 'r')
sys.stdout = open('output.txt', 'w')

# Leggi input
N = int(input())
K = int(input())
print(N + K)
```

Input di stringhe in C++

Per leggere una linea intera con spazi, usate `getline`. Attenzione: dopo aver letto con `cin`, dovete pulire il buffer con `cin.ignore()`!

```
int K;
string S;
cin >> K;
cin.ignore(); // pulisci il buffer!

for (int i = 0; i < K; i++) {
    getline(cin, S); // legge tutta la linea
    // fai qualcosa con S
}
```

Input di liste in Python

Per leggere numeri sulla stessa riga separati da spazi, usate `split()`

```
N = int(input())
distanze = input().split()
D = []
for i in range(N):
    D.append(int(distanze[i]))

# Ora D contiene tutti i numeri
```

In C++, `cin` si ferma al primo spazio o "*a capo*".
In Python, `input` si ferma al "*a capo*".

Output in C++

Per stampare array o vector su una riga, separati da spazi

```
vector<int> risultati = {0, 2, 1, 4, 3};
for (int i = 0; i < risultati.size(); i++) {
    cout << risultati[i] << " ";
}
cout << "\n";
// Output: 0 2 1 4 3
```

Output in Python

Per stampare liste su una riga, separati da spazi

```
risultati = [0, 2, 1, 4, 3]
for i in range(len(risultati)):
    print(risultati[i], end=" ")
print()
# Output: 0 2 1 4 3
```

`print` di base stampa sempre alla fine un "*a capo*". `end=" "` fa si che invece venga stampato solo uno spazio. Per alcuni esercizi, avrete bisogno di stampare tanti pezzi attaccati l'un l'altro: usate `end=""`.

Int & Floats in C++

- int: da `-2^31` a `2^31 - 1`, circa da `-2 * 10^9` a `2 * 10^9`
- long long: da `-2^63` a `2^63 - 1`, circa da `-2 * 10^18` a `2 * 10^18`
  - potete accorciarlo se scrivere in cima `typedef long long ll;`
  - `long long x = 3` --> `ll x = 3`
- float & double: numeri decimali, ma attenti ad errori di arrotondamento
  - `if(a == b)` --> `if(abs(a-b) < 1e-9)`

Int & Floats in Python

In Python gli interi possono essere grandi quanto vuoi!

```
x = 10**100  # nessun problema
```

Per i decimali:

```
a = 3.14
b = 2.71
# Attenti agli errori di arrotondamento
if abs(a - b) < 0.0000001:
    print("Uguali")
```

Modulo

Il modulo `%` è l'operazione che ritorna il resto di una divisione

`13 / 5 = 2` con resto di 3 --> `13 % 5 = 3`

- `(a + b) % m = (a % m + b % m) % m`
- `(a * b) % m = (a % m * b % m) % m`
- `(a - b) % m = (a % m - b % m) % m`
  - se minore di zero, aggiungete `m`

Modulo

```
print(13 % 5)  # 3

a = 100
b = 200
m = 7
print((a + b) % m)  # 6
print((a * b) % m)  # 6
```

Macros in C++

Potete abbreviare il vostro codice con le direttive `typedef` (per definire o accorciare un datatype), oppure abbreviare codice con le macro `#define`. Il compilatore lo scambierà con il codice originale.

```
#define PB push_back
#define FOR(i, n) for (int i = 0; i < n; i++)
typedef vector<int> vi;
int N = 10;
vi v; // vector<int> v
int main()
{
  FOR(i, N) { v.PB(i); } // for (int i = 0; i < N; i++) { v.push_back(i); }
  FOR(j, N) { cout << v[j]; } // for (int j = 0; j < N; j++) { cout << v[j]; }
}
```

Cicli in Python

Python non ha macro, ma i cicli sono più semplici:

```
N = 10
v = []

# Ciclo da 0 a N-1
for i in range(N):
    v.append(i)

print(v)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Stampare gli elementi
for i in range(N):
    print(v[i])
```

Variabili globali in C++

Per essere efficienti, salvare l'input globalmente vi permette di accederlo da tutto il codice. Salvarlo in un gigantesco array è comodo. Altri vantaggi sono che, ad esempio, un array globale viene inizializzato con tutti gli elementi a zero, mentre dentro il `main` no.

```
#DEFINE MAXN 1'000'000 // un milione
int N; // dimensione input
int a[MAXN]; // array lungo MAXN tutto di zeri
int main() {
  cin >> N;
  FOR(i, N) { cin >> a[i]; }
  FOR(i, N) { /* il tuo codice */ }
}
```

Variabili globali in Python

Anche in Python potete usare variabili globali:

```
N = int(input())
a = []

for i in range(N):
    numero = int(input())
    a.append(numero)

# il tuo codice
for i in range(N):
    print(a[i])
```

Stringhe in C++

In C++ possiamo salvare le stringhe come `string` anziché `char[]`, con più funzioni disponibili.

```
string s;
string s2 = "Hello";
int main() {
  cin >> s; // si ferma al primo spazio o al primo "a capo" (newline \n)
  getline(cin, s); // si ferma al primo \n
  // immaginiamo di inserire "World is beautiful" in s
  cout << s2 + s; // "HelloWorld is beautiful"
  cout << "lunghezza: " << s.size(); // 18 // va bene anche s.length();
  cout << s[0] << s[s.size() - 1]; // 'W' + 'l'
  cout << s.substr(9, 6); // (index iniziale, lunghezza) --> "beauti"
}
```

Stringhe in Python

Le stringhe in Python sono molto semplici:

```
s = input()  # legge una linea
s2 = "Hello"

# immaginiamo che s = "World"
print(s2 + s)  # "HelloWorld"
print(len(s))  # lunghezza: 5
print(s[0])  # 'W' (primo carattere)
print(s[2])  # 'r' (terzo carattere)
```

Vectors in C++

Array dinamici la cui lunghezza può variare. Possiamo aggiungere e togliere elementi a piacere in O(1) ammortizzato.

```
vector<int> v;
v.push_back(3); // inserisci in fondo --> [3]
v.push_back(2); // inserisci in fondo --> [3, 2]
v.push_back(5); // inserisci in fondo --> [3, 2, 5]

cout << v[1]; // 2
cout << v.size(); // 3

cout << v.back(); // ultimo elemento --> 5
v.pop_back(); // rimuove l'ultimo elemento. ma non lo ritorna --> [3, 2]
cout << v.back(); // ultimo elemento --> 2
```

Vectors in C++

```
vector<int> v2 = {1, 2, 3, 4, 5}; // inizializzato con questi valori
vector<int> v3(10);    // size = 10, tutti i valori = 0
vector<int> v3(20, 5); // size = 20, tutti i valori = 5
```

Per iterare

```
for (int i = 0; i < v.size(); i++) { // loop standard
  int a = v[i];
  cout << a << "\n";
}
for (auto a : v) { // loop automatico
  cout << a << "\n";
}
```

Liste in Python

Le liste sono l'equivalente dei vector in C++:

```
v = []
v.append(3)  # [3]
v.append(2)  # [3, 2]
v.append(5)  # [3, 2, 5]

print(v[1])  # 2
print(len(v))  # 3
v.pop()  # rimuove l'ultimo --> [3, 2]
```

Liste in Python

```
v = [1, 2, 3, 4, 5]
```

Per iterare:

```
for i in range(len(v)):
    print(v[i])

for numero in v:  # più semplice!
    print(numero)
```

Python e strutture dati

Rispetto a C++, in Python usiamo solamente liste e dizionari.

Python è un linguaggio più lento di C++, perchè favorisce un esperienza per developer più facile rispetto alla velocità di esecuzione più veloce. Mentre strutture dati come bitset o multiset sono ottimizzazioni per specifiche situazioni, su python questi non sono necessari.

Se programmerete in python, avrete più tempo per l'esecuzione del codice: la parte difficile ed importante rimane creare l'algoritmo efficente e corretto!

Set

Un `set` memorizza elementi unici in ordine crescente. È utile per evitare duplicati e avere gli elementi ordinati automaticamente.

```
set<int> s;
s.insert(3); // {3}
s.insert(1); // {1, 3}
s.insert(4); // {1, 3, 4}
s.erase(3);  // {1, 4}
if (s.find(4) != s.end()) { cout << "4 trovato\n"; } // find() ritorna l'iterator
else { cout << "4 non trovato\n"; }
cout << s.count(4) << " " << s.count(3); // "1 0"
cout << s.empty(); // false
```

Ordine dei Set

I `set` salvano gli elementi in modo ordinato. Per questo motivo, inserire e prendere costa O(log n) anziche' O(1)

```
for (auto it = s.begin(); it != s.end(); it++) {
  // it è un puntatore ad un elemento del set
  auto x = *it;
  cout << x << "\n";
}

for (auto x : s) {
  cout << x << "\n";
}
// solo se contiene 3, altrimenti abbiamo *s.end() che non esiste
cout << *s.find(3);
```

Set in Python

I set in Python sono simili:

```
s = set()
s.add(3)
s.add(1)
s.add(4)
s.remove(3)

if 4 in s:
    print("4 trovato")

print(len(s))  # 2
for numero in s:
    print(numero)
```

Map

Struttura `key-value`: data una chiave, ci viene ritornato il valore corrispondente. Un `array` è una `mappa` dove le chiavi sono numeri

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
cout << m["aybabtu"] << "\n"; // non esiste, viene creato con 0 e ritorna
if (m.count("aybabtu")) { /* key exists */ }
for (auto x : m) {
  auto key = x.first; auto value = x.second;
  cout << key << " " << value << "\n";
}
```

Dictionary in Python

I dizionari sono l'equivalente delle map in C++:

```
m = {}
m["monkey"] = 4
m["banana"] = 3
m["harpsichord"] = 9

print(m["banana"])  # 3

if "banana" in m:
    print("trovato!")

for chiave in m:
    print(chiave, m[chiave])
```

Multiset in C++

Un `multiset` tiene il conto di quante volte è stato inserito un elemento. Ha le stesse proprietà di un `set`

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3

s.erase(s.find(5)); // eliminane uno
cout << s.count(5) << "\n"; // 2

s.erase(5); // eliminali tutti
cout << s.count(5) << "\n"; // 0
```

Multiset in Python

In Python usiamo un dizionario per contare. Anche in C++ possiamo usare una mappa, ma il multiset è più veloce per alcune situazioni.

```
s = {}
s[5] = 0

s[5] += 1
s[5] += 1
s[5] += 1
print(s[5])  # 3

s[5] -= 1
print(s[5])  # 2
```

Iterators & Ranges in C++

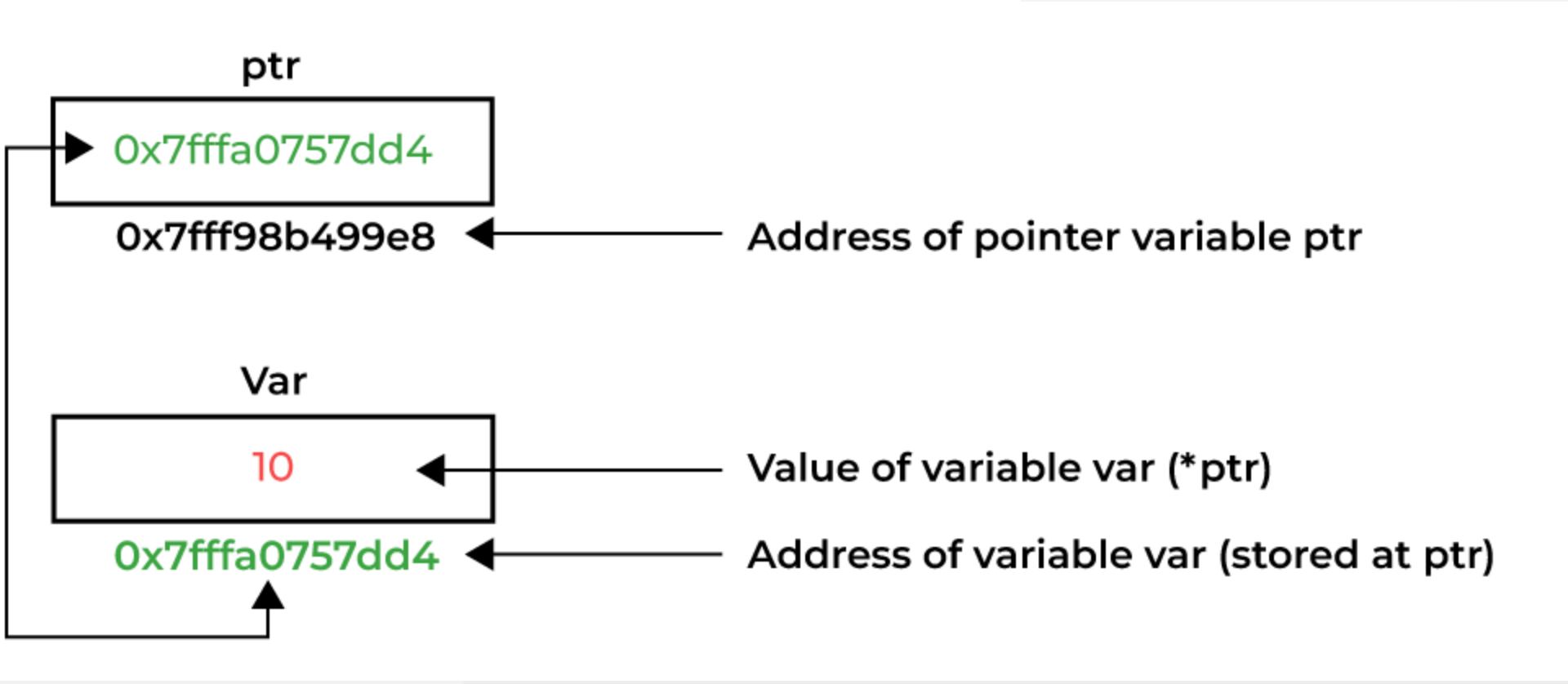
Cosa sono `v.begin()` o `s.end()`? Puntatori al primo e *dopo* l'ultimo elemento. Vuol dire che `*v.begin()` esiste, ma `*v.end()` no.

```
// alcune funzioni con gli iterator
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());

// con gli array semplici, un puntatore e' la variabile + index
// a[5] == *(a+5) == 5[a]
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Pointers

Un puntatore è una variabile che contiene l'indirizzo di un'altra.



Funzioni utili in Python

Python non ha puntatori, ma possiamo comunque usare delle funzioni molto comode

```
v = [5, 2, 8, 1, 9]

v.sort()
print(v)  # [1, 2, 5, 8, 9]

v.reverse()
print(v)  # [9, 8, 5, 2, 1]
```

Bitset

Array di 0 o 1, ma dove ogni valore è salvato come singoloo bit. Più efficente di un array di booleani, e possiamo usare operatori logici tra di loro

```
bitset<4> s; // [0000]
s[1] = 1; // [0100]
s[3] = 1; // [0101]
cout << s.count(); // conta gli 1, cioè qui 2

bitset<4> s2(string("0111")); // [1110], letto al contrario
cout << (a & b) << " " << (a | b); // 0100 1111
```

Esercizi assieme

- Quasi-Isogram (isogram)
- Destroy the Village (barbarian)