1 Linguaggio C++

Il linguaggio consigliato nelle gare di competitive programming è il C++. Questo è tra i più veloci disponibili, tra i più consolidati e implementa direttamente diverse librerie base. Questa guida non è un corso di programmazione in C++, ma verranno riportate cose che solitamente non vengono trattate.

1.1 Costrutti particolari

I costrutti del linguaggio che andremo a trattare in questo capitolo sono quelli che vengono solitamente tralasciati in quanto possono creare confusione in un programma normale che deve essere mantenuto. Non è il caso dei programmi nelle gare di informatica, quindi ci si può sentire più che motivati a utilizzarli.

break Questo costrutto permette di interrompere l'esecuzione di un ciclo e continuare dalla riga successiva alle parentesi grafe. Bisogna ricordare che per cicli intendiamo `for, while, do while`, non l'`if`; se viene usato all'interno di un `if`, questo prenderà in considerazione il primo ciclo che trova. È utile quanto troviamo una soluzione e non vogliamo inserire controlli nella condizione del ciclo. Rende il programma più facile da usare.

continue Simile al `break`, può essere usato solamente all'interno di cicli. Questo riprende l'esecuzione dalla verifica della condizione di ciclo. Anche questo può risultare utile in programmi in cui vogliamo fare una serie di operazioni e ritornare un elemento per il vettore.

auto Questa parola rappresenta il tipo automatico, ovvero quello che il compilatore può intuire in fase di compilazione. In generale è utilizzabile solo nel caso in cui l'assegnazione del valore è diretta nella dichiarazione ed esplicita, ad esempio come risultato di una funzione.

ragne for Questa operazione è possibile nei tipi iterabili, In particolare, ne è consigliato l'uso con i costrutti della standard library che vedremo in seguito. Quesrto prende in input una variabile di iterazione (costante, copia del valore), e un range; per range intendiamo un oggetto che ha gli attrivuti `begin()`, `end()` e il risultato di begin supporta `operator++`, oppure liste di elementi tra parentesi.

```
vector<int> v;
....
for(const int& e:v){
  cout<<e<<" ";
}
...
for(int e:{1,2,3}){
    cout<<e<<" ";
}
```

Notare che in molti casi è comodo usare la parola `auto` per assegnare la variabile, sopratutto se si parla di strutture complicate.

2 Lettura problema

In questa sezione daremo una prima occhiata a cosa consiste il testo di un problema delle gare di informatica. Il testo di una gara a squadre consiste nelle seguenti sezioni:

Intestazione Vengono indicate informazioni sulla gara (nome, data), e la **difficoltà** . Bisogna tenere conto di questo parametro, che ci indica quanto è risolvibile un problema in base alle nostre conoscenze.

Testo Qui è presente la descrizione del problema, dobbiamo leggere con attenzione e capire cosa richiede il problema.

Nota testo Vendono riportate informazioni più o meno utili, ad esempio dove trovare i file per eseguire l'input, e informazioni particolare sul problema (i tipi delle variabili)

Input Contiene come è formattato l'input. È consigliato di leggerlo solamente se non si usa il file di input id default, e per fare delle prove sul testo. Non è necessario controllare che il file sia formattato bene.

Output Contiene le informazioni per come formattare l'output. Notare che sia l'input che l'output vengono letti da terminale direttamente, non da file.

Constraints Queste sono le limitazioni del problema per ottenere punteggio pieno. In tutte le occasioni è necessario leggere e tenere presente che il problema possa ipoteticamente risolvere tutti i problemi. Ad esempio le dimensioni massime vanno prese da questa sezione.

Scoring In questa sezione sono riportati i testcase presenti e i punti ottenibili da ognuno di essi. Vanno letti se non si ha idea su come risolvere il problema in maniera efficiente, quindi si cerca se è possibile casi in cui risulta facile.

Le informazioni sui limiti dell'input e l'efficienza relativa degli algoritmi sono riportati nella sezione 2.1.

Esempi Sono riportati gli esempi di input e output. Prima di far valutare un programma è necessario verificare che il programma passi almeno questi.

Explanation Vengono riportate le spiegazioni su come una soluzione nei casi di input è stata trovata e si dimostra che è corretta. Da leggere se non si capisce cosa voglia il problema.

2.1 Tempi e complessità

Nei problemi di informatica la cosa importante che dobbiamo tenere presente non è l'ottimizzazione della singola istruzione, ma della complessità dell'algoritmo. Per passare i problemi nelle gare le soluzioni devono avere di solito i seguenti vincoli

Complessità	N	Esempi height
$O(n)$	$N < 10^7$	Ricerca massimo, operazioni singole
$O(n \log n)$	$N < 10^6$	Sorting, uso strutture ordinate
$O(n^2)$	$N < 3000$	DP con matrici
$O(2^n)$	$N < 20$	Brute force

Questi limiti sono molto indicativi, dato che dipendono anche dall'implementazione, ma in generale se un programma non li rispetta non passa il testcase.

3 Valutazione programma

Quando si ritiene di aver realizzato un programma che

4 Guida rapida al Competitive Programming in C++

Questa guida riassume le cose da sapere per competere nelle gare di informatica.

I concetti trattati sono riassunti, per informazioni, chiarimenti e modifiche, scrivere a matteo.canton2@gmail.com

4.1 Linguaggio C++

È necessario sapere le basi scolastiche del C++. In CP non è necessario scrivere codice elegante e bello, ma solo funzionale e rapido nella scrittura. È dunque autorizzato (e consigliato) l'uso di variabili globali, funzioni che terminano prima del momento, utilizzo di costrutti quali `continue`, `break`, `auto` ... .

Per compilare è consigliato usare il compilatore presente nel server (in OIS *g++*). Questo accetta diversi parametri, che è bene sapere esistere e essere utili. In particolare, i parametri consigliati sono `-Wall -fsanitize=address -O1 -g`. Questo permette di dire errori, mettere parametri di compilazione che danno informazioni in caso di *segment fault* (superamento dell'array). Si ricorda che avendo a disposizione un file, ed eseguendolo tramite `./a.out < input.txt` prende i dati dal file direttamente da `cin`.

Per il debug è consigliato in base al tempo e le conoscenze, l'utilizzo delle seguenti tecniche:

- log informazioni (scrivere i valori delle variabili). Questo è il classico metodo, ma si può fare elegante, ad esempio usare `clog` o `cerr` al posto di `cout`, che non viene visualizzato nell'output in fase di esecuzione sul server. Definire una macro di log è la cosa migliore. Si usa il comando tipo `#define log(x) (clog<<"["<< __LINE__ <<"]"<<#x<<" -> "<<x<<endl,`

`x)`. Questa funzione, scritta con `log(x);`, scrive un *stderr* la riga, l'espressione, il risultato, e lo ritorna. Si può usare anche tipo `foo(log(x))`.

- Uso debugger. Questi dipendono dall'IDE, sono più efficienti, ma spesso ci vuole più tempo.

Ricordarsi di tenere i file ordinati secondo qualche ordine, e fare copia incolla fa risparmiare tempo.

4.2 Libreria standard

Questa libreria è fondamentale da conoscere per ottenere risultati accettabili!

Questa libreria si include con l'inclusione di `#include <bits/stdc++.h>`. Questa libreria include tutte le librerie standard di c++, e funziona solo con g++. Tra le librerie viene incluso `algorithms,, pair, vecotor, stack, set, priority_queue, map,...`

Le librerie più importanti sono le seguenti:

- `algorithms`: include algoritmi più comuni da usare, quali `sort(begin, end,SortFunction)`. **DA USARE, NON IMPLEMENTARE IL SORT SEMPLICE**.
- `vector`: vettori dinamici di lunghezza variabile, si può aggiungere e togliere in fondo in $O(1)$
- `pair`: insieme di due elementi, chiamati `left` e `right`. È presente il confronto tra coppie confrontando prima il primo, poi il secondo.
- `stack, queue, dequeue`: classiche strutture
- `priority_queue`: Permette di ottenere il massimo da i numeri inseriti. Si può inserire, togliere, trovare il massimo, in $O(\log n)$.
- `set`: insieme di elementi ordinato, posso inserire, togliere, cercare, trovare successivo e precedente di un elemento, trovare i-esimo, in $O(\log n)$.
- `map`: come set, ma con possibilità di associare a un elemento un valore di un altro tipo

Tutte queste sono documentate nella documentazione sempre presente in gara. Non siate timidi a usare questo.

4.3 Grafi

Nei problemi capitano spesso i grafi. Questi si rappresentano in diversi modi:

Alberi con radice Vengono definiti per ogni nodo con il padre. È comodo rappresentare con un vettore di lunghezza n che contiene in posizione i la posizione del padre del nodo i. È indipendente partire da 0 o 1, quindi o ci si adatta a come vengono presentati oppure si fa sempre come si pensa. Possono essere rappresentati come alberi normali

Alberi Spesso li rappresentiamo come grafi semplici, e in caso creiamo una procedura per far diventare un nodo radice e costruire l'albero su questo.

Grafi Si rappresentano in due modi:

- Matrice di adiacenza: scarsa, NO USARE QUASI MAI (è n^2).
- `vector<int> G[MAXN]`. Un vettore che in posizione i contiene i nodi a cui è collegato il nodo `i`. Per costruire si fa nel seguente modo:

```
for(int i=0;i<n;i++){
  cin>>x>>y;
  G[x].push_back(y);  //creo collegamento da x a y
  G[y].push_back(x);  //vice versa
}
```

Se il grafo è pesato, si usa una pair, ovvero:

```
vector<pair<int,int>> G[MAXN];
\\...
for(int i=0;i<n;i++){
  cin>>x>>y>>w; // da x a y peso w
  G[x].push_back(make_pair(p,y));  //creo link da x->y
  G[y].push_back(make_pair(p,x));  //vice versa
}
```

Ovviamente si adatta al caso, tipo se è direzionato non aggiungo l'elemento x a $G[y]$.

Le procedure di base sono le seguenti:

```
// DFS: esplora prima in profondita',
//    per ogni vicino, vado su quelli vicini a questo prima.
int seen[MAXN];

vector<int> G[MAXN];
void DFS(int i){
  seen[i]=1;
  //prendo tutti i nodi che solo collegati a i
  for(auto e:G[i]){
    if(seen[e]==0){
      //posso fare cose e restituire cose in certe condizioni.
      DFS(i);
    }
  }
}

// visita per distanza, prima quelli vicini, poi quelli lontani.
void BFS(int i){
  seen[i]=1;
  queue<int> s;
  q.push(i);
  seen[i]=1;
  while(!q.empty()){
    i=q.pop_front();
    for(auto e:G[i])
    if(seen[e]==0){     //aggiungo alla coda gli altri nodi
```

```
27        seen[e]=1;
28        q.push(e);
29      }
30      //faccio cose sul nodo i
31    }
32  }
```

Esempi classici sono trovare il connettività, trovare percorso minimo, etc.. FARE ESERCIZI.

4.4 Trucchi per gare

Per le gare di informatica, è importante avere presenti i seguenti concetti

- Leggere i *testcase* è importante. Non è detto che si riesca a fare tutto, ma spesso è facile passare i primi testcase e prendere parecchi punti. Inoltre se il programma dice che viene dato in input un vettore lungo n e tra le specifiche ho $n \leq 10^7$, sono sicuro che posso dichiararne uno di questa dimensione, quindi faccio semplicemente `#define MAXN 10000008`, ovvero la lunghezza totale + 8 (che non dispiace, se ad esempio finisco per qualche caso finisco in posizione n). I vettori li dichiaro `int a[MAXN]`. Nella maggior parte dei problemi non è necessario tenere conto della memoria usata, eccetto rare occasioni (DP con matrici, strutture sparse). In tutti questi casi esiste comunque un metodo per ovviare al problema (utilizzo coppie di vettori, `map`).

- Il codice deve essere asintoticamente veloce, e abbiamo un compilatore veloce. Non serve ottimizzare nella singola variabile, o sul ricorsivo/iterativo. È più importante fare cose efficienti a livello asintotico, ovvero $O(n^2)$ è molto peggio di $O(n \log n)$, indipendentemente da quante righe scrivo e quanto eleganti e annidate sono.

- Capire i codice di errore è importante. In particolare, quando si fa il submit di un programma e risulta sbagliato è fondamentale capire quello che si sta sbagliando, per poter capire l'errore. I problemi più comuni sono i seguenti:

  - *Soluzione errata* Questo errore è il più comune e il più difficile da trattare. Il problema è che il nostro programma restituisce un valore sbagliato. È necessario dunque analizzare il codice e verificare cosa non torna.
  - *Soluzione fuori tempo massimo* Questo errore è dovuto al fatto che il nostro programma è troppo lento per trovare la soluzione ottimale richiesta. Anche in questo caso è necessario diminuirne la complessità e trovare gli eventuali colli di bottiglia.
  - *Memory violation, segmentetion foult*: sono errori di accesso in memoria, in particolare il programma accede a memoria non allocata oppure non accessebile. In questo caso bisogna capire cosa scatena il

problema (array troppo piccolo, accesso a locazioni successive del vettore, memory leak, dereference a null). La maggior parte delle volte il problema è legato all'uso scorretto scorretto degli indici o all'allocazione di un vettore che non rispetta i requisiti dati in base all'input del problema.

- *Errore di compilazione* Il programma non compila. È necessario verificare se l'ambiente utilizzato è simile a quello del compilatore sul server, e provare a compilare lo stesso codice. Solitamente è facile da risolvere.

Se siamo convinti che il programma funzioni (ad esempio non vanno i casi base), è possibile che abbiamo caricato il file sbagliato, sul problema sbagliato, oppure non abbiamo salvato. È bene tenere presente che anche fare pochi punti non è un problema, se non viene in mente una soluzione migliore è inutile perdere tempo a ragionare sullo stesso problema e restare bloccati.

- A volte i problemi usano numeri interi grandi, quindi è necessario salvarli come `long long int`. Per non perdere tempo a scrivere `long long int` ogni volta, nel file di intestazione da cui fare copia incolla scrivi tra le altre cose di debug e include la direttiva `#define lli long long int`.

- Se il programma non passa il test case per il tempo, ad esempio con un limite di 2 secondi termina in $2.024ms$, questo non significa che non è passato di poco, ma che è stato terminato prima, quindi va ottimizzato.

$$\sum_{i=1}^{5}\left((2n)^4-(2n-1)^4\right)=1-\sum_{i=1}^{10}i^4+\sum_{i=1}^{5}(2i)^4=$$