Introduzione alle gare di programmazione

Luca Chiodini

Aprile 2014

Indice

Premessa

Questa piccola guida è stata scritta avendo come obiettivo l'avvicinare gli studenti alle gare di programmazione. Nella mia esperienza ho notato come spesso vi sia un *gap* tra le conoscenze fornite, per esempio, da un corso di studi di tipo informatico a livello di scuola superiore e il bagaglio di strumenti e nozioni che si rende necessario durante una gara di informatica.
Non vuole questa guida essere un libro di strutture dati e algoritmi: su ciò si trovano testi in abbondanza. Questa piccola guida vuole bensì essere un utile ausilio a colmare questo divario dando consigli pratici e utili per rendere al meglio durante le gare.
In bocca al lupo!

Luca Chiodini, 2014
luca@chiodini.org

Capitolo 1

Sistema di correzione

Una delle sfide più interessanti della gestione di una gara di programmazione è la valutazione delle soluzioni sottoposte dagli studenti. In particolare, poiché gli utenti contemporanei sono spesso numerosi e non si può pensare che il giudice sia una persona umana, nel corso degli anni sono stati scritti diversi sistemi che implementano una valutazione automatica. CMS è quello adottato nella finale nazionale delle Olimpiadi Italiane di Informatica, nelle Olimpiadi Internazionali di Informatica e in numerose altre competizioni. È quindi bene che un atleta conosca, almeno in piccola parte, queste piattaforme.

1.1 Problemi durante la sottoposizione

Dopo la sottoposizione del sorgente assicurati innanzitutto che questa sia stata accettata. Alcune possibili spiegazioni di un rifiuto sono:

- Sottoposizione troppo ravvicinata alla precedente. Per evitare che il server si sovraccarichi, alcune gare consentono la sottoposizione di soluzioni solo dopo che un certo intervallo di tempo trascorso è trascorso rispetto all'ultimo invio. La soluzione in questo caso è semplice: aspetta un attimo prima di inviare la tua nuova soluzione.

- Limite massimo di sottoposizioni raggiunto. Per prevenire il *reverse engeneering* dei casi di input e il sovraccaricamento del server, alcune gare consentono solo un numero limitato di sottoposizioni per ogni problema. Assicurati di sapere se tale limite viene imposto e regolati di conseguenza.

- Formato di file non riconosciuto. Il server non ha riconosciuto l'estensione del file e non sa come comportarsi: assicurati che il file abbia un'estensione e che essa sia tra quelle consentite per quella gara.

1.2 Problemi durante la compilazione

Se si verifica un errore durante la compilazione assicurati innanzitutto di:

- avere sottoposto l'ultima versione che stavi modificando (e non un altro file, magari con nome identico, ma salvato in un altro percorso);
- avere verificato che l'errore non avviene anche durante una compilazione in locale.

Se nonostante queste verifiche il sistema non riesce a compilare la tua soluzione, una spiegazione potrebbe risiedere in qualche libreria mancante. Ricorda di includere *sempre* tutte le librerie necessarie (alcune versioni di DevC++ includono automaticamente `<stdlib.h>`) e che la piattaforma di correzione è basata su Linux dove quindi alcune funzioni presenti su Windows (come `system("pause")`) non funzioneranno mai.

1.3 Problemi durante l'esecuzione

L'esito per ogni singolo testcase può essere:

- Output corretto: il programma ha fornito la soluzione corretta.
- Output errato: il programma ha fornito una soluzione errata. Il caso tipico è un programma che "sbaglia" (ovvero che effettivamente produce un risultato diverso da quello previsto), ma in questa categoria potrebbe rientrare anche un programma con *Output malformato* (vedi punto seguente).
- Output malformato: il programma ha fornito una soluzione che non rispetta *formalmente* la tipologia di output richiesta per quel problema. Controlla che spazi e fine-riga corrispondano a quelli richiesti, che il numero di linee in output sia conforme a quanto richiesto, *eccetera*...
- Tempo massimo di esecuzione superato: il programma ha impiegato più del tempo consentito per quel problema. Nella quasi totalità dei casi, ciò è dovuto all'utilizzo di un algoritmo poco efficiente (leggi il capitolo sulla complessità computazionale).
- Esecuzione terminata a causa di chiamata di sistema vietata: il programma sta facendo uso di chiamate a syscall che la sandbox ritiene vietate. Rimuovi la parte di codice interessata.
- Esecuzione terminata a causa di accesso a file vietato: il programma stava tentando di leggere un file vietato e la sandbox lo ha interrotto. Rimuovi questa parte di codice.

- Esecuzione fallita a causa di codice di ritorno diverso da zero: il programma ha ritornato un valore diverso da 0 o `EXIT_SUCCESS`. Controlla il codice e modificalo affinché rispetti questa regola.
- Esecuzione terminata con un segnale: specialmente quando il segnale è l'11, ciò può indicare una violazione dei limiti di memoria (vedi il capitolo sulla complessità computazionale). Non allocare più memoria di quanto ti sia concesso.

Come puoi vedere sono molti gli esiti con cui un programma può terminare: sopra sono elencati quelli disponibili con CMS. In ogni caso, il sistema dovrebbe fornirti un'indicazione che ricalca quelle dell'elenco precedente.

Capitolo 2

Tipologie comuni di esercizi

2.1 I/O con stdio

La modalità "classica" di comunicazione di input e output è per mezzo di *standard input* (per intenderci, la tastiera) e *standard output* (il monitor). Se questo è il caso, in C++ la soluzione più semplice è utilizzare la classe `iostream` nel seguente modo:

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int a, b;
6    cin >> a >> b;
7    cout << a + b << endl;
8  }
```

Attraverso l'operatore `>>` si acquisisce in una variabile il valore inserito dall'utente (può essere un numero, un carattere oppure una stringa di caratteri fino allo spazio).
Analogamente, attraverso l'operatore `<<` si mostra a schermo una variabile (oppure un valore).
Nella quasi totalità dei casi, non c'è alcuna necessità di modificare il comportamento di `cout` perché automaticamente visualizza correttamente la variabile a seconda del tipo.
Nelle gare può occasionalmente esserti chiesto di stampare numeri con un certo numero di cifre decimali. Se questo è il tuo caso, il manipolatore `setprecision(n)` (per il cui utilizzo è necessario includere `iomanip`) imposta in modo fisso il numero di cifre decimali da visualizzare quando cout avrà a che fare con operazioni in floating point.
Nota come è anche possibile concatenare gli operatori `<<` e `>>` per mostrare e acquisire più valori in una sola istruzione.

2.2 I/O da file

Negli esercizi di tipo *batch* le operazioni di input e di output vengono tipicamente gestite tramite lettura e scrittura su file (avviene così, per esempio, nelle Olimpiadi Italiane di Informatica nella selezione territoriale). In C++ la soluzione più semplice è utilizzare la classe `fstream` (che, come suggerisce il nome, è simile a `iostream`) nel seguente modo:

```
#include <fstream>
using namespace std;
int main()
{
  int a, b;
  ifstream in("input.txt");
  in >> a >> b;
  ofstream out("output.txt");
  out << a + b << endl;
}
```

Il listato dichiara due oggetti (passando come parametro il nome del file), rispettivamente di input e di output, da utilizzarsi in modo del tutto analogo a `cin` e `cout`.

2.3 Realizzazione di una funzione (e compilazione con grader)

Un'altra possibile implementazione richiesta è scrivere, anziché un intero programma con il proprio *main*, solo una funzione. Essa tipicamente riceve i valori come parametri e deve:

- ritornare un valore

```
int somma(int a, int b)
{
  return (a + b);
}
```

- riempire un array passato come parametro

```
void sommaarray(int N, int *a, int *b, int *c)
{
  for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

Lo svantaggio di questa richiesta è che rende leggermente più complicata la compilazione in locale, perché è necessario compilare il grader fornito insieme al proprio codice. Fortunatamente puoi usare un IDE (Code::Blocks è uno tra i più semplici), creare un progetto e aggiungere entrambi i file sorgente. Per controllare da dove il grader prende i suoi valori che passa alla tua funzione, sarà sufficiente una rapida ispezione al suo codice.

2.4 Output only

In questa tipologia di esercizi non dovrai sottoporre al correttore il sorgente del codice che hai scritto, bensì ti verranno forniti direttamente i casi di input. Tutto ciò che dovrai fare sarà eseguire in locale il tuo programma (con I/O da file) e consegnare tutti i file di output al sistema di sottoposizione.

Capitolo 3

Complessità computazionale

3.1 Definizione di algoritmo

In tutte le gare è necessario scrivere *algoritmi*, spesso codificandoli in un linguaggio di programmazione, per risolvere l'esercizio proposto. Un *algoritmo* non è altro che una **sequenza finita di istruzioni elementari** che risolve un problema in un numero *finito* di passi.

3.2 Valutare un algoritmo

In generale si può assumere che, dato un problema, ci siano molti algoritmi risolutivi ugualmente corretti. Questo perché, di fronte a un problema, ogni persona ha un approccio diverso a seconda delle conoscenze che ha sull'argomento, dell'esperienza, *eccetera...*
Esistono però dei parametri che determinano quale algoritmo è *migliore* di un altro. Le due caratteristiche tipicamente misurate durante le gare sono il *tempo di esecuzione* impiegato e la quantità di *memoria* utilizzata. Uno dei problemi maggiori nel calcolare il *tempo di esecuzione* di un algoritmo è che esso varia, anche significativamente, a seconda di un gran numero di fattori:

- le caratteristiche hardware della macchina che fa girare l'eseguibile: un server dotato di una grande potenza computazionale, con processori magari cento volte più veloci di un computer domestico, inevitabilmente eseguirà il programma in un tempo ridotto;
- l'ottimizzazione del compilatore che trasforma il codice sorgente in linguaggio macchina: a seconda di quanto "bravo" è il compilatore a tradurre alcune istruzioni, l'eseguibile prodotto sarà eseguito più velocemente;[1]

[1] Si veda ad esempio il parametro -O di GCC.

- il carico del sistema su cui viene eseguito il programma: se esso è elevato, il sistema operativo può essere costretto a delle scelte che riducono le prestazioni dei singoli processi (in termini di CPU e di I/O) a favore della stabilità e dell'equità globale.

È quindi evidente che bisogna accordarsi su un formalismo condiviso per indicare *quanto tempo* impiega un algoritmo: si è soliti indicare il **numero di istruzioni** che devono essere eseguite in relazione all'input.
Vediamo un esempio per chiarire il concetto:

```
1    for (int i = 0; i < N; i++)
2        cin >> array[i];
```

Quante istruzioni *elementari* vengono eseguite?

- inizializzazione del contatore (`i = 0`);
- *N+1* confronti (`i < N`);
- *N* incrementi (`i++`);
- *N* letture da `stdio` e inserimento nella i-esima locazione del vettore (`cin >> array[i]`).

In totale vengono quindi eseguite $3N + 2$ istruzioni.

3.3 Notazione O grande

Nell'indicare la complessità di un algoritmo non è mai necessario contare le singole istruzioni: l'obiettivo è riuscire ad avere un idea di massima della bontà di un algoritmo e arrivare quindi a poter effettuare dei confronti sulle prestazioni.
Per fare ciò è stata introdotta la notazione O grande, che qui verrà trattata in modo quasi irriguardoso dal punto di vista matematico, ma comunque sufficiente per lo scopo.
Riprendendo l'esempio precedente, si può dire che la funzione $3N + 2$ è asintotica per $x \to +\infty$ a N a meno di costanti (in questo esempio, una costante moltiplicativa 3 e una additiva 2). Ciò significa che per valori *molto grandi* di x, la funzione iniziale può essere più semplicemente rappresentata tenendo conto solo del termine più "pesante".[2]
Seguono alcuni esempi di qual è la notazione O grande di alcune funzioni:

- $7N^2 + 5N = O(N^2)$
- $2N + logN = O(N)$
  N è un infinito di ordine maggiore rispetto a $logN$

[2]Formalmente, il termine il cui ordine di infinito è maggiore.

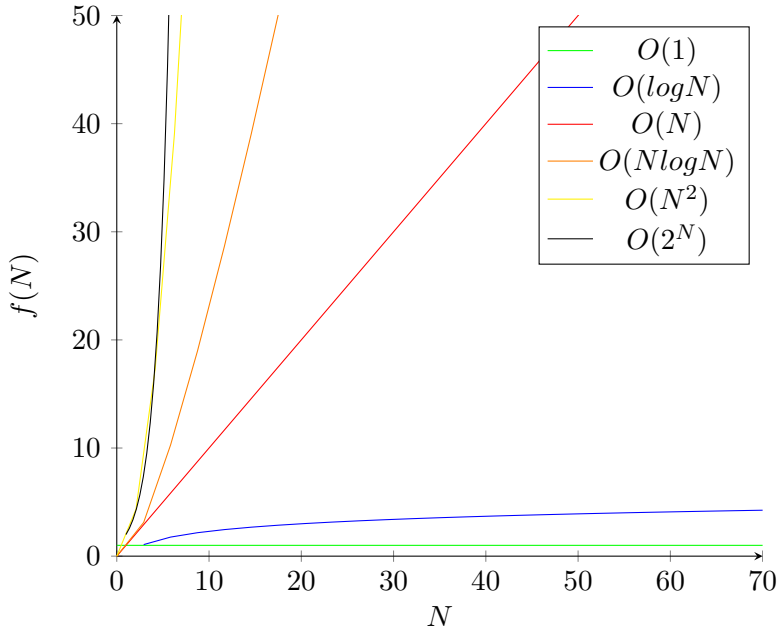
- $2^N + N^4 = O(2^N)$
  La funzione esponenziale è un infinito di ordine maggiore rispetto a qualsiasi funzione polinomiale (ovvero, del tipo x^k)

3.4 Classi di complessità computazionale

Per completezza si riportano *in ordine crescente* le classi di complessità computazionale che compaiono solitamente nelle gare.

Nome classe	Notazione O
Costante	$O(1)$
Logaritmica	$O(logN)$
Lineare	$O(N)$
Polinomiale	$O(N^k)$
Esponenziale	$O(k^N)$
Fattoriale	$O(N!)$

È importante citare anche $O(NlogN)$ che, sebbene non costituisca una classe a sè, compare frequentemente (è la complessità degli algoritmi di ordinamento "veloci").



3.5 L'importanza della complessità nelle gare

Se il discorso fatto nei paragrafi precedenti ti ha lasciato indifferente, questo paragrafo è stato scritto appositamente per farti cambiare idea.
Quando in informatica si definisce un algoritmo più **veloce** rispetto a un altro (per esempio $O(N)$ rispetto a $O(N^2)$), non si sta disquisendo su inezie (ovvero un programma che impiega qualche millisecondo in più); si discute piuttosto se un programma è in grado di terminare in tempi ridotti oppure se finirà nel giro di qualche millennio!
Facciamo un esempio concreto con il calcolo dell'n-esimo numero di Fibonacci: per definizione (ricorsiva)

$$F(n) = F(n-1) + F(n-2)$$

il che porta a un algoritmo *naïve* di complessità esponenziale $O(2^N)$.
Memorizzando i risultati intermedi in un vettore, è semplice scrivere un algoritmo che calcoli l'n-esimo numero di Fibonacci in tempo lineare ($O(N)$).
Proviamo ora a calcolare diversi numeri di Fibonacci con i due algoritmi e misuriamo i tempi di esecuzione:

N	alg. esponenziale	alg. lineare
10	$< 0.01s$	$< 0.01s$
40	$0.88s$	$< 0.01s$
100	$10^{18}s$ [3]	$< 0.01s$

Nota come già con numeri piccoli (100 lo è), l'algoritmo esponenziale non diventa solo *un po' più lento*: diventa **intrattabile**. 10^{18} secondi è **maggiore** del tempo trascorso da quando è nato l'universo!
Quando leggi il testo di un problema è quindi importante individuare subito l'ordine di grandezza massimo dell'input su cui verrà testato il tuo programma. Se per esempio un problema ha posto $N = 1000$, puoi stare tranquillo di non eccedere il tempo consentito anche con un algoritmo di tipo quadratico ($1000^2 = 1000000$, un numero di operazioni che un calcolatore è tranquillamente in grado di svolgere in molto meno di un secondo), mentre uno di tipo esponenziale sicuramente non totalizzerà il massimo del punteggio. Questa stima iniziale ti consente anche di non perdere tempo a scrivere algoritmi *più efficienti* di quanto serva: sempre nel caso di prima, sebbene scrivere una soluzione lineare può essere un buon esercizio, in una gara è probabilmente tempo sprecato che puoi invece dedicare ad altro.

[3] Ovviamente il tempo è solo una stima (ottenuta tramite proporzione, considerando 2^N come complessità).

3.6 Complessità spaziale

Con la stessa notazione O grande che abbiamo visto in precedenza, è possibile stabilire anche quanto *spazio in memoria* occupa un programma durante la sua esecuzione. Capita che alcuni esercizi pongano dei limiti di memoria: stai attento a non superarli! Senza addentrarsi troppo, sono sostanzialmente due le complessità riscontrabili (oltre a quella costante, dove il consumo è pressoché nullo):

- lineare ($O(N)$), quando si alloca ad esempio un vettore per contenere l'input:

```
#define MAXN 1000
int array[MAXN];
```

- quadratico ($O(N^2)$), quando si alloca ad esempio una matrice per una risoluzione con programmazione dinamica:

```
#define MAXN 1000
int dp[MAXN][MAXN];
```

Nella pratica, basta un semplice calcolo: `sizeof(tipo)*C` (dove `C` è la complessità: 1, N, N^2, *ecc...* e `sizeof(tipo)` restituisce quanti byte occupa una variabile di tipo `tipo`) deve essere minore del limite imposto.

Capitolo 4

Complessità di algoritmi e strutture

Quando avrai acquisito una certa competenza nell'uso delle strutture dati e degli algoritmi, potrà capitare che in una gara si ponga questo problema: so risolvere il problema nel modo A e nel modo B, quale dei due è più conveniente?
Dal capitolo precedente abbiamo imparato che tutto sta nel valutare quale algoritmo tra A e B ha una complessità computazionale inferiore.
È però *lento* contare a mente quante istruzioni verranno eseguite. Poiché però la complessità di alcuni algoritmi è nota a priori e ampiamente studiata, la strategia migliore consiste nello scomporre il problema e individuare gli algoritmi la cui complessità computazionale è nota.

4.1 Vettori e liste

I vettori esistono in due tipi: statici e dinamici. Per la trattazione che segue però verranno accomunati sebbene nella realtà vi siano alcune differenze di complessità (ve ne sono alcuni accenni nella documentazione della libreria STL).
La tabella riporta il comportamento delle due strutture in operazioni tipiche.

Operazione	Vettore	Lista
Allocazione	$O(N)$	$O(N)$
Inserimento nuovo elemento	$O(N)$[1]	$O(1)$
Eliminazione nuovo elemento	$O(N)$[2]	$O(1)$
Accesso all'i-esimo elemento	$O(1)$	$O(N)$
Accesso all'elemento successivo	$O(1)$	$O(1)$

Calcolando quanto frequentemente è richiesto di accedere ad elementi *sparsi* e quanto inserire o eliminare degli elementi, è facile stabilire se la struttura dati appropriata al problema è un vettore o una lista.

4.2 Ricerca e ordinamento

Ricerca e ordinamento sono due operazioni che capitano piuttosto frequentemente ed è importante sapere la loro complessità.
In generale, esistono due tipi di ricerca:

- ricerca lineare: partendo dall'inizio del vettore, lo scorro finché trovo il valore cercato. Evidentemente questa operazione è lineare ($O(N)$).
- ricerca binaria (o dicotomica): dividendo ogni volta a metà la parte di vettore in cui cercare, la complessità è logaritmica ($O(log_2 N)$). Ricorda che questo tipo di ricerca è applicabile *solo* se il vettore è ordinato.

Dal grafico presente nel capitolo sulla complessità computazionale puoi facilmente desumere che già per N modesti il secondo tipo di ricerca è molto più veloce.

N	$O(N)$	$O(log_2 N)$
10	10	4
1000	1000	10
1000000	1000000	20
1000000000	1000000000	30

Se i dati vengono forniti già ordinati, la tabella mostra che non c'è ragione per non usare la ricerca binaria.
Il dubbio può sorgere quando i dati sono sparsi e sono eventualmente da ordinare: meglio *ordinare* e poi usare la ricerca binaria oppure meglio usare semplicemente la ricerca lineare?
Molto dipende anche da quale algoritmo viene impiegato per ordinare: gli algoritmi più *naïve* richiedono $O(N^2)$ (BubbleSort è tra questi), mentre altri come QuickSort impiegano solo $O(N log_2 N)$.
Proviamo a fare due conti, considerando K il numero di valori da cercare e N la dimensione del vettore:

1. se adottassimo l'algoritmo di ordinamento "lento" e la ricerca binaria, la complessità sarebbe

$$O(N^2) + K * O(log_2 N)$$

[1] Shift verso destra.
[2] Shift verso sinistra.

2. se adottassimo l'algoritmo di ordinamento "veloce" e la ricerca binaria, la complessità sarebbe

$$O(Nlog_2N) + K * O(log_2N)$$

3. se decidessimo di non ordinare e di usare la ricerca lineare, la complessità sarebbe

$$K * O(N)$$

È chiaro quindi che non c'è una risposta univoca alla domanda "cosa è meglio fare?", perché la complessità finale dipende molto da K (ovvero da quante volte dobbiamo cercare).
Poniamo K a tre valori significativi e calcoliamo le complessità finali[3]:

K	Soluzione 1	Soluzione 2	Soluzione 3
1	$O(N^2)$	$O(Nlog_2N)$	$O(N)$
N	$O(N^2)$	$O(Nlog_2N)$	$O(N^2)$
N^2	$O(N^2log_2N)$	$O(N^2log_2N)$	$O(N^3)$

Per $K >= N$, ovvero quando il numero di ricerche è almeno uguale al numero di elementi in cui cercare, la soluzione che prevede ordinamento in $O(Nlog_2N)$ e ricerca in $O(log_2N)$ si rivela la soluzione migliore. Di contro, la tabella mostra chiaramente che non conviene ordinare l'intero vettore nel caso in cui poi si debbano cercare solamente pochi ($< N$) elementi.

4.3 ADT comuni

Oltre ai tipi di strutture dati "tradizionali", esistono anche delle strutture dati cosiddette *astratte* che spesso si mappano direttamente ad alcuni dei problemi proposti durante le gare: per quanto non siano strettamente indispensabili per prendere qualche punto, spesso lo sono per arrivare alla soluzione ottimale richiesta.
Poiché questo non è un libro di DS&A[4], verranno affrontate le ADT più comuni solo dal punto di vista della loro complessità computazionale e del loro utilizzo tipico.

4.3.1 Pile e code

Pile e code possono essere viste come un'estensione delle liste, con il vincolo aggiuntivo che è possibile solo un *accesso limitato* ad esse.

[3]Se alcuni risultati nella tabella ti sembrano strani, riguarda il capitolo sulla complessità computazionale.

[4]Strutture dati e algoritmi.

È importante sottolineare che **tutte** le operazioni su di esse (push, pop, size[5]) hanno complessità costante ($O(1)$).

4.3.2 Grafi

I grafi sono un ambito *enorme*, verranno qui trattati solo per fornire consigli per il loro utilizzo in una gara.
Ci sono due modi principali per memorizzare un grafo in memoria:

- liste di adiacenza, preferibili nei casi in cui il grafo sia sparso (ovvero quando ci sono *pochi*[6] archi)
- matrici di adiacenza, da adottare quando il grafo è denso oppure quando si vuole rispondere molto spesso e in tempo costante alla domanda: "C'è un arco tra i nodi a e b?"

Un argomento importante è la visita dei grafi. Se conosci già questa struttura, saprai sicuramente che esistono due modi per visitare tutti i nodi di un grafo: *in ampiezza* (BFS) oppure *in profondità* (DFS). Entrambe le modalità hanno una complessità $O(V + E)$ (dove V è il numero di nodi e E il numero di archi), quindi sostanzialmente lineare.
In ultimo è fondamentale citare l'algoritmo di Dijkstra che cerca i cammini minimi in un grafo. La sua complessità varia a seconda dell'efficienza della struttura dati nel fornire il minimo tra un insieme di dati: l'implementazione classica costa $O(V^2)$, mentre quella che fa uso di un *Fibonacci heap* costa $O(E + Vlog_2V)$.

4.3.3 Alberi

Gli alberi sono un tipo struttura non lineare che simulano la struttura di un albero "vero", con una radice e dei nodi.
Le operazioni tipiche definite sugli alberi binari *bilanciati* sono:

- visita completa (in qualsiasi ordine) in $O(N)$
- inserimento di un nuovo elemento in $O(log_2N)$
- eliminazione di un elemento in $O(log_2N)$
- ricerca di un elemento in $O(log_2N)$

Nota come tutte le operazioni tipiche hanno tutte complessità *logaritmica*.

[5] È costante solo con l'uso della libreria STL.
[6] Più formalmente, *sparso* significa che il numero degli archi è molto minore del quadrato dei nodi.

Capitolo 5

Le librerie STL e algorithm

5.1 Vantaggi nell'uso delle librerie

Capita, a volte, che qualche studente preferisce scrivere da sé il codice per alcune operazioni (per esempio, la ricerca binaria e l'ordinamento). Questo approccio ha almeno due potenziali problemi durante le gare:

- se è facile scrivere del codice "a memoria" quando si è da soli e in tranquillità, più difficile è solitamente farlo durante una gara quando la pressione della competizione e il tempo che inesorabilmente scorre possono portare a errori.
- spesso si impara il codice più *facile* da memorizzare, che però quasi mai coincide con il codice più *efficiente* in termini di complessità computazionale (vedi il capitolo dedicato).

Cerca di regolarti affinché, durante le gare, la tua difficoltà non stia nell'implementare *algoritmi noti*, ma eventualmente nel risolvere il problema in sé. Per fare ciò, è consigliabile imparare l'utilizzo delle librerie standard: STL e algorithm sono due tra le più utili.

5.2 La libreria STL

STL, Standard Template Library, è la libreria che definisce strutture dati generiche (ADT), iteratori e algoritmi.
In particolare, le più comuni ADT definite sono: vettori, liste, insiemi, code, pile e array associativi.
La documentazione completa della STL può essere trovata facilmente online[1]; comunque nei paragrafi successivi troverai un esempio di codice che ne chiarisce il funzionamento base.

[1] https://www.sgi.com/tech/stl/

5.2.1 Classe vector

La classe vector fornisce un vettore dinamico pronto all'uso.

```
1 #include <vector> // include la classe
2 int main()
3 {
4     vector <int> valori(3, 100); // alloca un vettore di interi,
          inizializzato con 100-100-100
5     valori.push_back(50);  // inserisce un nuovo elemento in coda
6     cout << valori.size(); // stampa la dimensione del vettore
7     for (vector<int>::iterator it = valori.begin(); it != valori.end(); it
          ++)
8         cout << *it << endl;
9 }
```

La notazione più strana è quella usata alle righe 7-8 per fare un ciclo che stampi tutti gli elementi nel vettore, che usa un nuovo concetto: gli **iteratori**. Per semplificare, puoi pensare a un iteratore come a un puntatore a una cella di memoria.
Nel caso di esempio `it` viene inizialmente inizializzato all'inizio del vettore `valori` (con `valori.begin()`) e viene incrementato ad ogni ciclo spostandosi sull'elemento successivo fino a che non finisce "fuori" dal vettore (`valori.end()`).
Per visualizzare il valore "puntato" da `it`, si usa la notazione `*it`. Sebbene inizialmente questa notazione può sembrarti ostica, con la pratica la scoprirai essere molto flessibile.

5.2.2 Classe list

La classe list realizza una lista doppiamente concatenata (ciò significa che si può scorrere avanti e indietro di una posizione sempre in $O(1)$).

```
1 #include <list> // include la classe
2 int main()
3 {
4     list <int> lista;      // inizializza una lista vuota di interi
5     lista.push_back(5);    // inserisce in coda 5
6     lista.push_front(10);  // inserisce in testa 10
7     cout << lista.front(); // visualizza l'elemento in testa
8     lista.pop_front();     // rimuove l'elemento in testa
9     for (list<int>::iterator it = lista.begin(); it != lista.end(); it++)
10        cout << *it << endl;
11 }
```

5.2.3 Classi stack e queue

Le classi stack e queue realizzano rispettivamente una pila e una coda.

```
#include <stack> // include la classe
#include <queue> // include la classe
int main()
{
    stack <int> pila; // inizializza una pila vuota di interi
    stack <int> coda; // inizializza una coda vuota di interi
    pila.push(5); // inserisce 5 nella pila
    coda.push(5); // inserisce 5 nella coda
    coda.push(8); // inserisce 8 nella coda
    while (!coda.empty()) // finche' la coda non e' vuota
    {
        cout << coda.front(); // stampa il primo elemento
        coda.pop();           // e lo rimuove dalla coda
    }
    while (!pila.empty()) // finche' la pila non e' vuota
    {
        cout << pila.top(); // stampa il primo elemento
        pila.pop();         // e lo rimuove dalla pila
    }
}
```

5.3 La libreria algorithm

Il mondo della programmazione si regge su strutture dati e algoritmi. Mentre la libreria STL, vista nel paragrafo precedente, fornisce delle affidabili strutture dati; la libreria algorithm fornisce un insieme di algoritmi per operare su di esse.
In particolare, è definita la funzione `sort` che realizza un ordinamento "veloce" ($O(nlog_2n)$), da utilizzare come di seguito.

```
#include <algorithm> // include la libreria
#include <vector>
int main()
{
    vector <int> vettore(10);
    for (int i = 0; i < 10; i++)
        cin >> vettore[i];
    sort(vettore.begin(), vettore.end());
}
```

Una chiamata a `sort` con solo due parametri (inizio e termine di un vettore di interi), ordinerà il vettore in ordine crescente.
Realizzando una funzione di confronto tra due elementi è però possibile usare `sort` per ordinare con il criterio voluto un vettore di un tipo qualsiasi. Per esempio:

```
bool confronta(int a, int b)
{
```

```
3     return (b < a);
4 }
5 int main()
6 {
7     vector <int> vettore;
8     sort(vettore.begin(), vettore.end(), confronta);
9 }
```

La funzione `confronta` deve ritornare *vero* se il primo parametro deve venire, nella sequenza ordinata, **prima** del secondo. Nell'esempio l'effetto è ordinare un vettore di interi in ordine decrescente.
Un'altra operazione comune risolvibile con le funzioni messe a disposizione dalla libreria è la ricerca binaria (o dicotomica).
La funzione `binary_search` restituisce semplicemente *se* un elemento è presente in una sequenza, mentre `lower_bound` e `upper_bound` sono utili per poterne cercare la posizione.

```
1 #include <algorithm> // include la libreria
2 #include <vector>
3 using namespace std;
4 int main()
5 {
6     vector <int> vettore(10);
7     for (int i = 0; i < 10; i++)
8         cin >> vettore[i];
9     // Esempio di utilizzo della binary_search per cercare 5
10    if (binary_search(vettore.begin(), vettore.end(), 5))
11    {
12        cout << "Nel vettore e' presente un 5" << endl;
13        vector<int>::iterator posizione;
14        posizione = lower_bound (vettore.begin(), vettore.end(), 5);
15        // La posizione e' la differenza rispetto all'inizio
16        cout << "5 trovato in posizione " << posizione - vettore.begin();
17    }
18    else
19        cout << "Nel vettore non e' presente un 5" << endl;
20 }
```