Competitive Programming

Davide Cazzin

Novembre 2021

1 Complessità

Un concetto molto importante nel Competitive Programming è quello della complessità di un algoritmo. Gli algoritmi che scriviamo devono essere efficienti sia in termini di tempo che di spazio. Spesso i problemi che dovremo risolvere ci daranno un indicazione sulla dimensione dei dati in input, il tempo massimo per il quale possiamo eseguire il nostro programma e la memoria massima che può utilizzare.

Vediamo 3 classi di complessità, big O, big theta (Θ) e big omega (Ω).

- Big O: Complessità asintotica, complessità dell'algoritmo nel peggior caso possibile. Definisce un limite superiore sulla complessità dell'algoritmo.
- Big theta: Se big O e big omega sono uguali allora si può dire che big theta è uguale a big O e big omega.
- Big omega: Complessità dell'algoritmo nel miglior caso possibile. Definisce un limite inferiore sulla complessità dell'algoritmo.

Le classi di complessità possono essere utilizzate per descrivere sia la complessità temporale che la complessità spaziale di un algoritmo.

2 Esempio complessità

Vediamo ora un esempio di complessità di alcuni algoritmi.

Listing 1: Esempi complessità

```
// Restituire la media di un vettore di interi
// Complessità Theta(n)
double media(const vector<int>& v) {
    if(v.empty()) return 0;
    int somma = 0;
    for (int i = 0; i < v.size(); i++) {
        somma += v[i];
    }
    return somma / (double) v.size();
}

// Restituire l'indice del primo numero negativo del vettore, -1 se non esiste
// Complessità O(n) e Omega(1)
int firstNegativeNumberIndex(const vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
```

```
        if (v[i] < 0) return i;
    }
    return -1;
}

// Restituire l'elemento più piccolo in un vettore di interi contentente almeno un elemento
// Complessità Theta(n)
int minElement(const vector<int>& v) {
    int res = INT_MAX;
    for (int i = 0; i < v.size(); i++) {
        res = min(res, v[i]);
    }
    return res;
}

// Restituire il più piccolo intero in un vettore ordinato in modo non decrescente e contenente almeno un elemento
// Complessità Theta(1)
int minElementInSortedVector(const vector<int>& v) {
    return v[0];
}

// Stampare a video il nome dei tre colori RGB
// Complessità Theta(1)
void printRGBColors(){
    vector<string> colors = {"red", "green", "blue"};
    for(const string& color : colors){
        cout << color << endl;
    }
}
```

2.1 Bubble Sort

Supponiamo di dover scrivere l'algoritmo Bubble Sort per ordinare gli elementi di un vettore in ordine non decrescente.

Listing 2: Bubble Sort in C++

```
#include<iostream>
#include<vector>

using namespace std;

void bubbleSort(vector<int>& v){
    bool ordered = false;
    unsigned int i = 0;
    while(i < v.size() && !ordered){
        ordered = true;
        for(unsigned int j = 0; j < v.size() - i - 1; j++){
            if(v[j] > v[j + 1]){
                int temp = v[j];
```

```
                v[j] = v[j + 1];
                v[j + 1] = temp;
                ordered = false;
            }
        }
        i++;
    }
}

int main(){
    vector<int> v = {1,5,3,2,7,4,6};
    bubbleSort(v);
    for(const int& el : v){
        cout << el << "\n";
    }
}
```

La compessità temporale di questo algoritmo è $T(n) = n * \frac{n}{2}$, diremo quindi che la classe di complessità è $O(n^2)$, il peggior caso si verifica quando il vettore è ordinato con ordine decrescente. Il miglior caso si verifica quando il vettore è già ordinato in ordine crescente, se tale condizione si verifica il vettore verrà visitato una sola volta e poi l'algoritmo terminerà, possiamo quindi dire che $\Omega(n)$.

2.2 Binary Search

Vediamo la complessità dell'algoritmo Binary Search. Il Binary Search è un algoritmo di ricerca che serve per trovare efficientemente un elemento in un vettore ordinato.

Listing 3: Binary Search in C++

```
#include <iostream>
#include <vector>

using namespace std;

int binarySearch(vector<int> v, int x){
    int low = 0;
    int high = v.size() - 1;
    while(low < high){
        int mid = (low + high) / 2;
        if(v[mid] == x){
            return mid;
        } else if(v[mid] < x){
            low = mid + 1;
        } else{
            high = mid - 1;
        }
    }
    return -1;
}

int main(){
```

```
    vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    cout << binarySearch(v, 5) << endl;
}
```

Per calcolare esattamente la complessità di questo algoritmo esistono diversi metodi matematici, tuttavia non ci soffermeremo troppo su questo. Nel competitive programming è importante saper calcolare velocemente la complessità.

Ad ogni iterazione del ciclo while, il numero degli elementi che stiamo considerando viene dimezzato. Continuiamo il ciclo while fino a che o troviamo l'elemento oppure finiamo gli elementi (se low >= high). Ad esempio supponiamo che gli elementi siano 7 e non l'elemento che stiamo cercando non sia presente nel vettore.

$\lfloor 7/2 \rfloor = 3$

$\lfloor 3/2 \rfloor = 1$

$\lfloor 1/2 \rfloor = 0$

La funzione matematica che calcola il numero di volte che un numero è divisibile per 2 è la funzione logaritmo, in questo caso il logaritmo in base 2 di 7.

$log2(7)$

Diciamo quindi che la complessità dell'algoritmo Binary Search è $O(log(n))$ e $\Omega(n)$.

3 Sort

In competitive programming molto difficilmente dovremmo scrivere un algoritmo di sorting, la maggioranza delle volte useremo il sort che ci viene fornito dalla standard library.

```
void sort (RandomAccessIterator first, RandomAccessIterator last);

void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Il sort utilizzato da questa funzione è chiamato IntroSort e ha una complessità temporale $O(nlog(n))$ e spaziale $O(log(n))$.

Il comparator di default che viene utilizzato dalla funzione sort, nel caso non ne venga fornito un altro, è $less < int >$ e ordina gli elementi presenti all'interno della struttura dati in ordine non decrescente. Per ordinare il vettore con ordine non crescente possiamo usare il comparator $greater < int >$, come nel seguente esempio:

```
vector<int> v = {1,4,5,6,3,2};
sort(v.begin(), v.end(), greater<int>);
```

Nel caso volessimo ordinare il nostro vettore secondo un criterio diverso da quello del comparator $less < int >$ e $greater < int >$ allora dobbiamo definire un comparator. Supponiamo ad esempio di aver un vettore di $pair < string, int >$ e vogliamo che il vettore sia ordinato in base all'intero con ordine non decrescente e nel caso di interi uguali devono essere ordinati in ordine alfabetico.

Listing 4: Custom Comparator

```
/*
    Data una lista di cognomi di studenti e la loro relativa media,
    ordinali con ordine non crescente in base alla loro media,
    in caso di parità ordinali in ordine alfabetico in base al cognome.

    Input:
    n (numero di studenti)
```

```
    [cognome] [media]
    ...

    Output:
    [cognome] [media]
    ...

    Esempio di input:
    4
    Verdi 19
    Bianchi 27
    Rossi 28
    Gialli 27

    Esempio di output:
    Rossi 28
    Gialli 27
    Bianchi 27
    Verdi 19
*/

#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

bool sortByAscendingIntAndDescendingString(const pair<string,int>& a, const pair<string,int>& b){
    if(a.second == b.second) return a.first > b.first;
    return a.second > b.second;
}

int main(){
    vector<pair<string,int>> v;
    int n;
    cin >> n;
    string name;
    int avg;
    while(n--){
        cin >> name >> avg;
        v.push_back({name, avg});
    }
    sort(v.begin(), v.end(), sortByAscendingIntAndDescendingString);
    for(const pair<string,int>& el : v){
        cout << el.first << ' ' << el.second << "\n";
    }
}
```

4 Makefile

Listing 5: Makefile example

```
.PHONY: release

release: SortPairs.cpp
	g++ -Wall -O3 -std=c++17 --pedantic -o SortPairs sortPairs.cpp

debug: SortPairs.cpp
	g++ -Wall -g -std=c++17 --pedantic -o SortPairsDebug sortPairs.cpp

clean:
	rm -f SortPairs SortPairsDebug
```